Security Types for Web Applications

Antoine Delignat-Lavaud

Inria Paris – Rocquencourt
Advisors: Karthikeyan Bhargavan & Sergio Maffeis

March 16 - August 31, 2012

Context and state of the art

The exponential development of web applications, accelerated by the convergence of mobile and desktop platforms towards the web as illustrated by the Windows 8 apps, written in JavaScript and reusable from the PC to the smartphone, has led to a simultaneous increase in web attacks, which are relatively easy to find and exploit but can impact huge amounts of users, as was demonstrated by the recent and massive password leaks from Yahoo, LinkedIn and Last.fm.

After the hype caused by "Web 2.0", there has been a significant effort of the research community to create formal tools to protect webpages from potentially harmful JavaScript code loaded from external sources, such as a dishonest advertising company. This concept of Mashup security has led to development of new tools to analyse the security of JavaScript code, most notabely ADSafe and JSLint. However, these efforts have often been hampered by the dynamic nature of JavaScript, which makes it a language poorly suited to static analysis.

The studied problem

Starting from an overview of the security of host-proof web applications and their (mis-)use of cryptography, we investigate classes of web attacks that can be automatically discovered with the help of formal analysis tools. We focus our attention on the relatively neglected problem of loading trusted code in an untrusted heap environment shared by different origins, some of which may be malevolent.

Our contribution

As a starting point, we conducted a review of various host-proof web applications: in that design, the user's sensitive data is stored in encrypted form on the cloud for backup and synchronization purposes, and is only decrypted client-side using a password-derived key that stays unknown to the server.

Our results showed that such services, despite their popularity and security minded design, suffered from various vulnerabilities, most of which were easy to discover and exploit. We then addressed the question of loading trusted code into an untrusted environment, the dual of the Mashup problem.

We discover that even in a heap completely controlled by a malicious party, it is possible using language primitives to load code that is able to compute cryptographic primitives and keep a secret hidden from the attacker. We isolate a small but still expressive subset of JavaScript with this property and show that it is possible using typing techniques to check the defensiveness of an arbitrary input code.

Relevance of our results

We implemented our type system and proceeded to edit a popular JavaScript cryptography library to use our restricted JavaScript subset until it was accepted by our typechecker. We then used this defensive implementation of cryptography to build new primitives, such as secure communication between cross-origin principals in an unsafe environment. We also compared the performance of our transformed code with the original. However, our work still lacks an important component: a formal proof of the security guarantees of defensive JavaScript in an untrusted heap.

Our paper reviewing web attacks on host-proof applications was accepted at the Workshop on Offensive Technologies for the 2012 Usenix Security Symposium. During our stay in Seattle for the conference, we met members of the F* team at Microsoft Research who had been working on connected problems in the context of fully abstract compilation of ML into JavaScript and whose result we hope can help us achieve our goal of proving the formal security of our subset.

Conclusion and further research

The last step required to complete this work is without a doubt to back our defensiveness claim with a proven security theorem in arbitrary heaps in a semantics that can account for even the most nefarious JavaScript features, such as setters in the base object prototypes.

This work will be carried on in the form of a PhD at INRIA Paris.

Master's Thesis
Inria Paris – Rocquencourt
March – August 2012

Advisors: Karthikeyan BHARGAVAN, Sergio MAFFEIS

Security Types for Web Applications

by

Antoine DELIGNAT-LAVAUD
antoine@delignat-lavaud.fr

Contents

Foreword

This report presents the results of a 20-week internship of the author at Inria Paris, under the supervision of Karthikeyan Bhargavan, head of team PROSECCO (Programming Securely with Cryptography) and Sergio Maffeis from Imperial College during the author's stay in London and his visit at Inria. Since neither is a native French speaker, and some of the material in this report comes from a paper published at the Workshop on Offensive Technologies (WOOT) from the 2012 USENIX Security Symposium, it is written in English.

1 Introduction

Since the broad popularization of Ajax, a web application design that relies on JavaScript to perform asynchronous requests to retrieve and handle dynamic data, there has been a clear trend towards moving application and user data from personal devices to the web, amplified by the rapid development of mobile platforms and an undeniable media hype for "Web 2.0" and "Cloud Computing".

While this evolution was permitted by the adoption and implementation of new standards (ECMA 262, DOM, XHTML, CSS), the security model of web browsers, centered around the same origin policy [1], itself based on domain names, has seen very little change until recently, when it became obvious that almost every big website or web application relied on a wealth of external libraries, content providers, advertisers and gadgets whose interactions could lead to severe security faults.

This resulted in a spark of interest of the research community leading to a variety of new defensive mechanisms:

- extensions of the browser's security policy: permission model for Google Chrome extensions [2], sandboxing techniques (HTML5 sandboxes, MashupOS [3], content security policy [4] to isolate code from data);
- information flow analysis (FlowSafe [5], Jif [6], jsflow [7]);
- restricted subsets of JavaScript in which security properties can be proven (ADSafe [8], Google Caja [9], FBJS);
- formal models of web applications (WebSpi [10], Alloy [11]).

To better understand the goals of these techniques, we now summarize the permission model of a modern web browser, which is schematized in figure 1.

When browsing `a.com`, a new JavaScript heap and DOM (Document Object Model, which consists of the abstract tree of the parsed HTML markup of the page and an API to walk and perform operations on it) are created.

`<script>` tags can be used to load JavaScript, either from `a.com` or any other origin like `ads.google.com`. Such scripts are evaluated in the same heap as the page, where it is possible to perform asynchronous requests, but normally only to `a.com` (in other words, it is possible to load cross-origin scripts but not to perform cross-origin Ajax requests. New mechanism have been introduced to bypass this restriction if the target server explicitely permits it).

`<iframe>` tags can be used to load external HTML from any origin, for instance the Facebook like button, into the page. Such frames always run in a separate JavaScript heap, and if the origin of the frame is different from its parent's, like in the Facebook case, the browser will isolate the subtree of the main DOM rooted at the `iframe`. This prevents the page from loading the Facebook login page and stealing the user's password by reading it from the DOM. However, this separation is not completely tight: communication by string messages is permitted via the `postMessage` method.

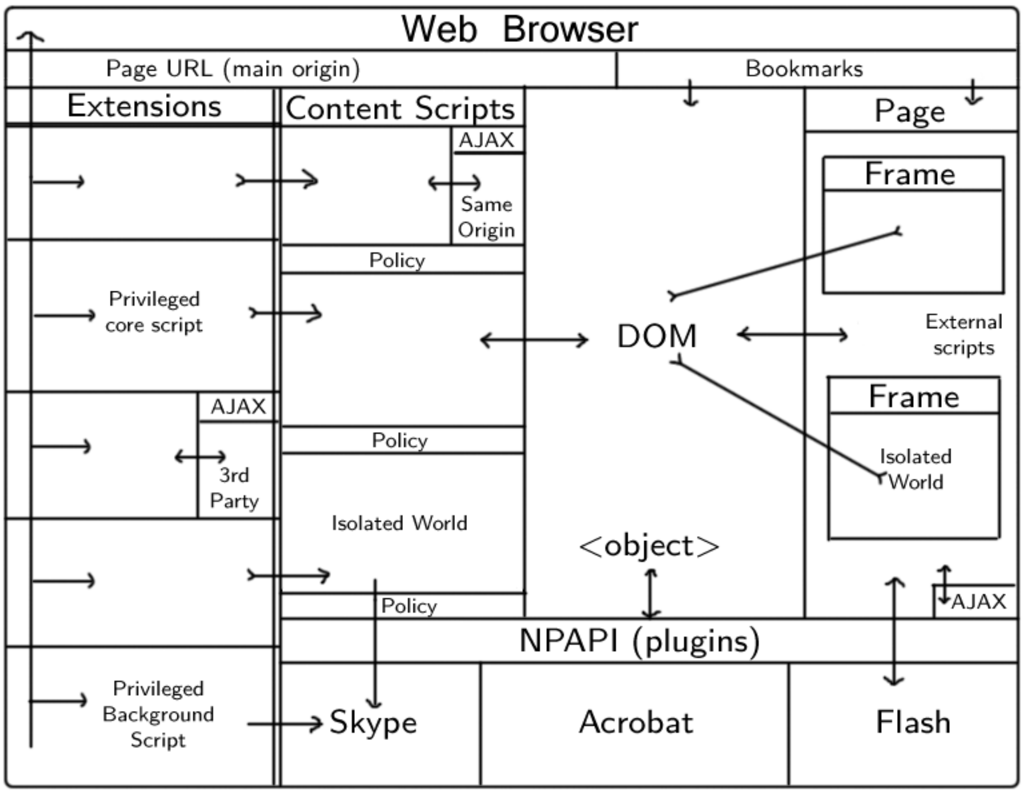

Figure 1: Main security principals in a modern browser

`<object>` tags can be used to interact with binary plugins running on the browser, for instance Flash or Java. These objects often rely on an input file (a JAR package in the case of Java), which is normally not permitted to be loaded across origins. Nevertheless, there are ways around this restriction, for instance to allow Youtube videos to be loaded on any page, leading to a history of vulnerabilities.

Additionally, there may be extensions running on the browser, which can request the permission to load content scripts on pages that match some origin patterns. Content scripts have access to the DOM of the page they are loaded into, but they run in a separate heap. Furthermore, they have the same origin restriction on Ajax requests as their host page and they don't have access to the privileged browser API of the core extension scripts. Still, they can communicate with them using a string message function.

Finally, bookmarks can also contain JavaScript code that gets loaded into the current page when clicked on by the user. They share the same heap as the host page.

The two most common classes of vulnerabilities in web applications are cross-site scripting (XSS) and cross-site request forgery (CSRF). The former happens when script from an untrusted origin gets access to the DOM and heap of the victim origin. It can then steal sensitive data from the DOM (e.g. password form fields) or from the victim's server using Ajax and leak it to the attacker using the fact that many HTML tags such as `<img>` or `<script>` allow cross-origin queries. This class of attack is very frequent due to the lack of code and data separation in HTML: virtually every tag can cause some inline JavaScript to be executed. Hence, improperly sanitized user input can end up in the DOM of the application, for instance in a comments section. The recentely introduced content security policy (CSP) allows website to disable all inline script evaluation in supporting browsers at the cost of enforcing best practise for code separation.

On the other hand, CSRF attacks rely on the fact that most websites use cookie based sessions to authenticate users. Cookies are origin-bound persistent data stored by the browser on the server's request and sent back along with every following query to that origin. Typically, they contain a session token that proves the user is logged into that website. Using an `iframe`, an attacker can then force the user to perform a malicious action, such as sending him to `changepassword.php?newpassword=xxx`. Such attacks are also extremely widespread, and require specific protections in the form of an action token or checking the `Referer` header to protect against.

2 Motivating Case Study

The first step of our work was to conduct an overview of a class of web applications relying on the host-proof design. The philosophy of this design is to leverage the extended capabilities of current browsers to store sensitive user data remotely in encrypted form and have every operation on the data be performed client-side using the user's decryption key, which remains unknown to the server.

Several reasons motivated this choice: firstly, all these applications are intended to protect sensitive information, such as personal files and passwords, making them prime targets for attacks; secondly, they are popular, commercial services which we expected to be very security concious;

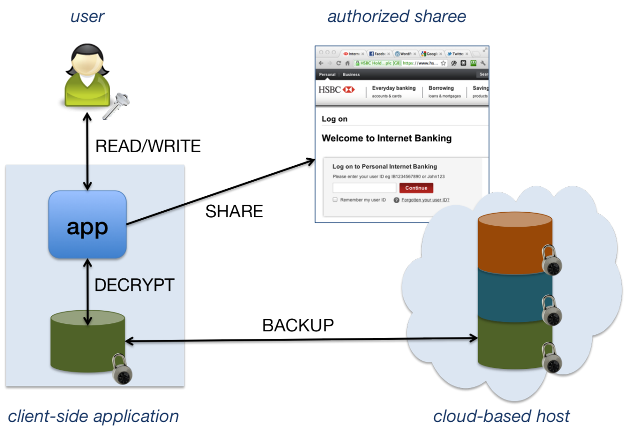

Figure 2: Host-proof web application architecture

lastly, all relied on client-side implementations of cryptographic primitives, notably in a JavaScript context prone to web attacks, a model that aside from WebSpi [10] is rarely considered by the formal analysis community.

The goal of this survey was to determine possible patterns of vulnerabilities and if tools could be built to automatically find and and defend against them. Most of the material from this section is from our paper at WOOT'12 [12], which is available from the conference website[1].

2.1 Host-Proof Web Applications

A host-proof web application follows the architecture depicted in Figure 2. 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 encryptions and decryptions, 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

[1]`https://www.usenix.org/conference/woot12`

the cleartexts 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) [13]. The design is also sometimes misleadingly called *zero-knowledge* [14, 15]. 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 [16] and SpiderOak [15], 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. Application such as BoxCryptor and CloudFogger add client-side encryption to non-host-proof storage services such as Dropbox.

- Password managers, such as LastPass [17], 1Password [18], and Roboform [19], offer to store users' confidential data, such as login credentials to different websites, or credit card 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. However, their common security goals are:

- *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.

We found five exemplary attacks on commercial host-proof applications that break these security goals by exploiting flaws in both their cryptographic design and their web interfaces. Tables 1 and 2 shows a brief summary of the features implemented by a number of host-proof applications. In particular, Table 1 notes the cryptographic algorithms and mechanisms used, while Table 2 summarizes the various web interfaces offered. We describe them in more detail when needed to explain our attacks.

2.2 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 paper support the PBKDF2 password-based key derivation function [20] that takes a passphrase p, salt s, and iteration count c, and generates an encryption key k (of a given length):

$$k = \text{KDF}(p, s, c)$$

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 [21]. 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 [22]. The attacks in this paper

Name	**Format**	**Key Derivation**	**Encryption**	**Encrypted Data**	**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: Example host-proof web applications and their cryptographic features

Name	**Backup Location**	**Remote Access**	**Bookmarklet**	**Client**	**Local Page**	**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 2: Example host-proof web applications and their web interfaces

do not rely on brute-force attacks against passwords. In the rest of this paper, we 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:

$$e = \text{ENC}(k, d)$$

The applications in this paper 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. In this paper, 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 passphrase-derived key)

$$h = \text{HMAC}(k', \text{ENC}(k, d))$$

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.

$$e = \text{ENC}(\text{HASH}(d), d), \text{ENC}(k, \text{HASH}(d))$$

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 all these cases, there is often no integrity protection. Moreover, some metadata, such as website URLs, may be left unencrypted to enable search and lookup.

When metadata is left unencrypted 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 attacken 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.

Towards Authenticated Encryption It is generally accepted among the cryptographic community that "encryption without integrity-checking is all but useless"[23]. A simple fix to tampering attacks would be to use an HMAC to protect the integrity of both the metadata and the encrypted items.

More generally, many host-proof applications use encryption algorithms as if they guaranteed ciphertext integrity. This is an incorrect assumption for many modes of AES and other algorithms. Instead, each password manager should seek to implement a scheme that provides authenticated encryption with associated data [24], where the associated data includes metadata such as website URLs.

Vulnerability Response We notified both 1Password and Roboform about these attacks on

April 3, 2012. The 1Password team responded within days that they are designing a new keychain format with integrity protections for their next version (4.0). The Roboform team proved more resistant. They first 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 currently anticipate any fixes. Both vulnerabilities have been publicly disclosed [25, 26].

2.3 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 [27]. 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 attack can be prevented by simply adding standard CSRF protections to all the JSONP URLs offered by SpiderOak. A more general design flaw is the management of encryption 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 [28], 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, which leaves shared folders on SpiderOak still open to other website attacks. Notably, many of the problems reported on the SpiderOak Security Response page have been cross-site scripting weaknesses.

2.4 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 [28]. 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 reconstructs and decrypt a complete copy of the user's directory tree. 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 key material and cryptographic operations in separate processes from web interfaces.

Vulnerability Response We notified the Wuala team about the vulnerability on May 21, 2012. They responded immediately and releaded an update (version 399) within 24 hours that disabled directory browsing on 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 [29].

2.5 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 but notably not on the user, but on the browser extension.

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 [30], and leading websites often get it wrong [31]. Perhaps the most widely used URL parsing library for JavaScript is parseUri [32] 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. 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 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 any eavesdroppers. We advocate that password managers implement site-specific authorization policies that include protocols and full domain names, enabling users to choose their desired level of security.

Vulnerability Response We notified 1Password about this vulnerability on April 3, 2012. The 1Password team responded within days and released a new beta version of their browser extensions (build 39304) that implements a new, more careful, URL parsing functions. This 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 [33].

2.6 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 [34] 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 [34] 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 unchanging master key 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 where an attacker can steal a page-specific, temporary encryption key 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 risks of leaking the master decryption key to the host page and changed their bookmarklet design within 24 hours. Decryption is now performed inside an iframe within the Lastpass origin, preventing the host page from stealing the key.

2.7 Case Study Conclusions

Our survey revealed different classes of problems in the design and implementation of host-proof applications; for each one, existing methods can be developed to address the issue:

- many applications had key management problems, amplifying the effects of web attacks, and other misuses of cryptographic primitives. In most cases, such issues are impossible to fix after the application has been deployed, hence it is highly desirable to encourage the use of formal analysis tools such as ProVerif [35] in the WebSpi model to verify the soundness of cryptographic protocols used by the application;

- access control problems and accidental release of private data can be detected with the progress of control flow and taint analysis techniques;

- JavaScript is poorly suited to implementing cryptosystems that are not vulnerable to web attackers. When decryption operations are performed in an untrusted host's heap, or in the case of decryption running along cross-origin scripts, there are many way for a malicious principal to spy on the decryption keys and plaintexts. Even though there has been a large amount of research in language-based security of mashups [8, 36, 37], they all aim to protect a trusted host page from untrusted external scripts (e.g. potentially malicious ads). The question of running trusted code in a potentially untrusted environment or communicating between two cross-origin scripts without the host (intentionally or inadvertently) tampering is not covered by existing research.

In the next section, we present our main contribution: a subset of JavaScript that is safe to run in an untrusted (rootkited) heap. We then demonstrate and implement an algorithm to decide membership to that subset, which we use to build an implementations of a hash function and encryption algorithm and how to use them to solve the above problem.

3 Defensive JavaScript Subset

The starting point of this section is the following problem: what are the security properties that can be enforced on code loaded into an arbitrary JavaScript heap? The LastPass bookmarklet from the previous section is an instance of this problem. This bookmarklet contains a secret shared with the LastPass server and used to prove that it is the user (and not the page on its own behalf) that is trying to log in. However, this secret is leaked to the host page when the bookmarklet is used, a problem that we found in every other bookmarklet we checked that contained a secret, like SuperGenPass, a bookmarklet able to generate domain specific passwords from a master password that if leaked to the host can reveal the user's password on other websites.
A first possible security property to consider would be to require that the heap, regardless of its initial state, remains unchanged after running a defensive script. However, this prevents any interaction with the host page, making the script effectively useless. Instead, we mark two heap locations: one for the page-specific parameters (stored as a primitive type: number, string or boolean) and a fresh one for releasing the result of the script to the page, we then require that the evaluation of the defensive script never accesses any heap location beside those two.
Even though this may seem like a simple property to prove, it turns out that there are many JavaScript features that make it very difficult to hold true, let along prove. Our initial goal was to prove this property in the operational semantics for JavaScript proposed by Maffeis [38], however some features from the latest ECMA specification of JavaScript that are very important to the security of defensive code are missing from the semantics. Hence, we do not yet have a formal security theorem for our JavaScript subset. Instead, we informally describe the possible attack vectors that we had to take into account when designing our language. We will return to this topic at the end of the section.

3.1 Possible Attack Vectors

Source code leak Assume that a secret is stored as a local variable inside a function: `function(){var secret = "x"; ...}`. If the host page gets a pointer to this function (for instance as the return value), it can steal the secret by type-casting the function to string, an operation which returns the source code of the function.

Call stack leak If an exception flows to the host page, it can (in some browser) stack information including source code fragments that can potentially contain secrets. Hence, defensive code that uses exceptions must be enclosed in a global `try` block.

Library function replacement Every library object and function in JavaScript, like `encodeUri`, `Date` or `Math` can be replaced or removed by the host page and cannot be used. Only language constructors and operators are available: defensive code must be fully self-contained.

Implicit function calls Because of dynamic typing, there are many operations in JavaScript that can trigger an implicit type cast with the potential to implicitely call user-defined functions: if `o` is an object, `o+''` will implicitely call the `toString` method of `o`. If it is not defined, the prototype chain is traversed until one is found. Hence, if the attacker redefines `Object.prototype.toString`, it can spy on the contents of `o` at the moment of the typecast.

Scoping Because a defensive script is not allowed to access its host heap, it is necessary to ensure that every variable access is local. This can be difficult in JavaScript: the `with(o)` statement adds to the current scope every (possibly dynamically created) property of `o` to the scope. To avoid this problem the use of `with` is restricted to objects for which the complete set of properties is known. Another problem is the `this` keyword, whose binding is completely dynamic and can refer to the global scope, to the containing object of a method, or to the instance of a created object. Finally, another feature breaks lexical scoping: functions have access to an `arguments` property which in turns may contain a pointer to a caller function and its arguments.

Prototype poisoning The 5th revision of the ECMA 262 standard introduced getters and setters, which are special properties that will call user specified functions when the property is accessed for reading or writing. If an attackers sets a setter using:

```
Array.prototype.__defineSetter__('0',
 function(v){alert('Got␣'+v); this[0]=v});
```

and the defensive script tries to initialize the first element of an initially empty array, it will reveal the current and all subsequent values of that element to the attacker. However, if the array was initialized with a first element: `a = [0]`, it is safe to use `a[0]` because the prototype chain is not traversed if the property exists in the object or array. The same issue exists for uninitialized object properties.
To avoid this attack, every array and object must be initialized literally, and there must be no access to undefined properties or array elements using the bracket operator.

3.2 Syntax Description

```
⟨defensive-program⟩ ::= ⟨statement⟩*

⟨statement⟩ ::= ε
  | 'with(' ⟨object_literal⟩ ')' ⟨statement⟩
  | 'if(' ⟨expression⟩ ')' ⟨statement⟩
    ('else' ⟨statement⟩)?
  | 'while(' ⟨expression⟩ ')' ⟨statement⟩
  | '{' ⟨statement⟩* '}'
  | ⟨expression⟩
  | ⟨statement⟩';' ⟨statement⟩

⟨expression⟩ ::=
  | ⟨expression⟩ @binop ⟨expression⟩
  | @unop ⟨expression⟩
  | ⟨lhs_expression⟩ @assignop ⟨expression⟩
  | ⟨expression⟩ '?' ⟨expression⟩ ':' ⟨expression⟩
  | ⟨lhs_expression⟩

⟨lhs_expression⟩ ::=
  | @identifier | ⟨literal⟩
  | ⟨lhs_expression⟩ '(' (⟨expression⟩ ',')* ')'
  | ⟨lhs_expression⟩ '[' ⟨literal⟩']'
  | ⟨lhs_expression⟩ '.' @identifier
  | ⟨function⟩

⟨function⟩ ::=
  | 'function' @identifier?
    '(' (@identifier ',')*'){'
    ('var' @identifier ('=' ⟨expression⟩)?)*
    ⟨defensive-program⟩
    ('return' ⟨expression⟩)? '}'

⟨object_literal⟩ ::=
  | '{' ( @identifier ':' ⟨expression⟩ ',')* '}'
  | '[' (⟨expression⟩ ',')* ']'
  | @number | @string | @boolean
```

3.3 Type System

Because of the imperative nature of JavaScript, we opt for a simple monomorphic type system. For simplicity and familiarity we describe the type system and inference algorithm in the Hindley–Milner style, but there is no universal type quantifier or generalization.

$\langle\tau\rangle$ ::=	number | boolean | string | undefined	
|	$\alpha,\ \beta$	Type variable
|	$\tilde{\tau} \to \tau$	Arrow
|	$\tilde{\tau}[\rho] \to \tau$	Method
|	$[\tau]_n$	Final Array
|	$[\tau]_{\geqslant k}$	Array schema
|	$\rho *$	Final object
|	ρ	Object schema

$\langle\rho\rangle$ ::= $\{l_1 : \tau_1, \dots, l_n : \tau_n\}$

During the inference phase, for instance of **function**(x){**return** x.a+x.b}, the object schema of x is allowed to be extended to $\{a : number, b : number\}$, whereas {a:0,b:1}.c will cause a unification failure because the literal object has a final type. Objects can also contain methods, which are functions that can refer to other properties in the object in which they are defined using `this`. The method signature has to match the type of the object where it is defined, for instance:

```
{a: function(){return this.b+this.c}, b:0,
    c:1}
```

We now present the main typing rules:

$$\text{ArrLit}\ \frac{\tau = fresh() \qquad \forall 1 \leqslant i \leqslant n, \Gamma \vdash e_i : \tau_i \quad \mathcal{U}(\tau, \tau_i)}{\Gamma \vdash [e_1, \dots, e_n] : [\tau]_n}$$

$$\text{ObjLit}\ \frac{\tau = \{\} \qquad \forall 1 \leqslant i \leqslant n, \quad \Gamma \vdash e_i : \tau_i \qquad \mathcal{U}(\tau, \{l_i : \tau_i\}) \quad \text{if } \tau_i = \tilde{\tau}_1[\rho] \to \tau_2 \text{ then } \mathcal{U}(\tau, \rho)}{\Gamma \vdash \{l_1 : e_1, \dots, l_n : e_n\} : \tau *}$$

We also need separate rules for methods and normal functions to deal with `this`:

$$\text{Fun}\ \frac{\begin{array}{c}\text{body} = (\text{var } y_1 = e_1, \dots y_m = e_m; s; \text{return } r) \\ \lambda = \text{fresh}() \qquad \tilde{\alpha} = \text{fresh}() \\ \forall j \leqslant m, \Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, (y_i : \mu_i)_{i<j} \vdash e_j : \mu_j \\ \Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu} \vdash s : \text{undefined}; r : \tau_r \\ \mathcal{U}(\lambda, \tilde{\alpha} \to \tau_r)\end{array}}{\Gamma \vdash \text{function } f(\tilde{x})\{\text{body}\} : \tilde{\alpha} \to \tau_r}$$

Compared to the function rule, methods have an additional object schema for `this`: $\rho = \{\}$, and after f, the binding `this` : ρ in the typing environment. Finally, the resulting type is $\tilde{\alpha}[\rho] \to \tau_r$. Reading object properties and array elements is easy:

$$\text{PropR}\ \frac{\tau = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \mathcal{U}(\{l : \tau\}, \sigma)}{\Gamma \vdash e.l : \tau}$$

$$\text{ArrR}\ \frac{\tau = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \mathcal{U}([\tau]_{\geqslant n+1}, \sigma)}{\Gamma \vdash e[n] : \tau}$$

However, function calls are potentially dangerous when calling methods outside objects. For this reason the call method rule, made of the composition of PropR and Call, is distinct.

$$\text{Call}\ \frac{\tau = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \Gamma \vdash \tilde{f} : \tilde{\alpha} \qquad \mathcal{U}(\tilde{\alpha} \to \tau, \sigma)}{\Gamma \vdash e(\tilde{f}) : \tau}$$

$$\text{MCall}\ \frac{\mu = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \mathcal{U}(\{l : \mu\}, \sigma) \qquad \Gamma \vdash \tilde{f} : \tilde{\alpha} \qquad \mathcal{U}(\tilde{\alpha}[\sigma] \to \tau, \mu)}{\Gamma \vdash e.l(\tilde{f}) : \tau}$$

Typing of expressions is straightforward, with monomorphic variants of the polymorphic operators, like + for the base types. Note that there is one typecast that is always safe to perform: the cast to boolean:

$$\text{BoolCast}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash !e : \text{boolean}}$$

Expressions are always typed `undefined` but they can require to typecheck an expression:

$$\text{While}\ \frac{\Gamma \vdash e : \text{boolean} \qquad \Gamma \vdash s : \text{undefined}}{\Gamma \vdash \text{while}(e)s : \text{undefined}}$$

Final type of SHA library object

```
{HASH:(string)[{SECRET:string, STR2BA:(string)
      => number array[>=50]}] => string, HASHED
     :boolean, LOCATION:string, SECRET:string,
     STR2BA:(string) => number array[50]} final
```

In its current form, there is a problem with the `with` rule. This rule adds an object's properties to the typing environment, but some of those properties may contain methods which no longer can be called with the call rule. In our implementation, we added an additional flag to the functions to lookup and add bindings to the environment to take this case into account.

$$\text{With}\;\frac{\Gamma \vdash e : \{\tilde{l} : \tilde{\tau}\}* \qquad \Gamma, \tilde{l} : \tilde{\tau} \vdash s : \text{undefined}}{\Gamma \vdash \text{with}(e)s : \text{undefined}}$$

The unification algorithm has some special traits specific to this type system. First of all, it causes side effects on type variables (as one would expect), but also on array and object schemas:

$$\begin{aligned}\mathcal{U}([\tau]_n, [\sigma]_m) &= [\mathcal{U}(\tau, \sigma)]_{\max(n,m)} \\ \mathcal{U}(\{\tilde{p}_1\}, \{\tilde{p}_2\}) &= \{\mathcal{U}(\tilde{p}_1 \cap \tilde{p}_2), \tilde{p}_1 \Delta \tilde{p}_2\}\end{aligned}$$

Furthermore, unifying a final object with an object schema consist in making sure the final object has all the properties required by the schema with compatible types. Similarly, unifying a final array with an array schema means checking that the types are compatible and that the final array has at least as many elements as required by the schema.

3.4 Implementation

The first step of implementing our design was to write a complete JavaScript parser using Menhir and ulex. Although such parsers already existed, notably in `js_of_ocaml` and the `LambdaJS` [39] implementation, they were missing important features such as unicode support and automatic semicolon insertion which prevented them from accepting real world JavaScript code.
We then implemented the type system using destructive type variables for unification. Since the subset presented above does not have dynamic array and object accessors, we extended the syntax to accept some constructions that include dynamic checks to ensure the safety of the property access:

$\langle dyn\_accessor \rangle$ ::=
| ($\langle x \rangle$ = @identifier) '`[`' $\langle expression \rangle$
'`&`' @posint '`%`' $\langle x \rangle$ '`.length ]`'
| @identifier '`[`' $\langle expression \rangle$ '`&`' @posint '`]`'

the first one ensures that the index to access is a positive integer between 0 and `a.length-1`. The second is especially short and efficient when the array to access is of length a power of 2.
Associated typing rules are straightforward:

$$\frac{\Gamma \vdash x : [\tau]_{\geqslant 1} \qquad \Gamma \vdash e : \text{int} \qquad n \in \mathbb{N}*}{\Gamma \vdash x[e\&n\%x.length] : \tau}$$

$$\frac{\Gamma \vdash x : [\tau]_{\geqslant n} \qquad \Gamma \vdash e : \text{int} \qquad n \equiv 0[2]}{\Gamma \vdash x[e\&n] : \tau}$$

Using these, we were able to implement SHA-256 and AES-256 CBC, based on code from the Stanford JavaScript Crypto Library (SJCL). Because of the literal declaration restriction, they can only operate on inputs of an arbitrarily chosen length. Cryptographic code is generally easy to make defensive: every array and object has to be given a complete initial value, dynamic array accessors have to be rewritten to include our dynamic bound check and some error handling and input validation code can simply be removed thank to the type system.
The final type of the SHA and AES library objects are given below. The SHA function only takes the first 50 bytes of the input into account, while this AES implementation works on 3 blocks.

Performance We measured the effect of making the SJCL implementation of AES-256 defensive. Given the dynamic checks added for bound

```
for(i=0; i < N; i++)
{
  a16 = M[i & N];
  for(t=0; t < W.length; t++)
  {
   W[t & 256 % W.length] = t<16
     ? a16[(t&256)%a16.length]
     : and32(sigma1(
       W[(t-2)&256 % W.length]
        )
       + W[(t-7)&256 % W.length]
       + sigma0(W[(t-15)&256 % W.length])
       + W[(t-16)&256 % W.length]))
  }

  a = H[0]; b = H[1]; c = H[2]; d = H[3];
  e = H[4]; f = H[5]; g = H[6]; h = H[7];

  for(t=0; t < W.length; t++)
  {
   T1 = h + Sigma1(e) + Ch(e, f, g)
        + K[t&512 % K.length] + W[t];
   T2 = Sigma0(a) + Maj(a, b, c);
   h = g; g = f; f = e;

   e = and32(d + T1);
   d = c; c = b; b = a;
   a = and32(T1 + T2);
  }

  addvm(H, [a,b,c,d,e,f,g,h]);
}
```

Figure 3: Main loop of SHA-256

conditions, one may expect to see a performance decrease, but there also has been some overhead reduction due to the removal of the input and error checks which are now performed statically. Bencharks were run on a 3Ghz Core i7 PC under Windows 7. Tested browsers are Chrome 21, Firefox 14, Internet Explorer 9 and Safari 5.1. Result is the average on 200 runs of the encryption of 20000 blocks, expressed in MB/s.

Browser	Chrome	Firefox	IE	Safari
SJCL	8.0	18.6	2.5	9.2
Defensive	16.4	24.0	2.6	9.2

```
function(input, dir)
{
  var key = this.key[dir & 1],
        a = input[0] ^ key[0],
        b = input[(!dir ? 1 : 3) & 3] ^
            key[1],
        c = input[2] ^ key[2],
        d = input[(!dir ? 3 : 1) & 3] ^
            key[3],
       a2 = 0, b2 = 0, c2 = 0, i = 0,
          kIndex = 4,
      out = [0, 0, 0, 0],
    table = this.Stables[dir & 1],
       t0 = table[0], t1 = table[1], t2 =
          table[2],
       t3 = table[3], sbox = table[4];

  for (i = 0; i < 13; i++)
  {
   a2 = t0[a>>>24 & 255] ^ t1[b>>16 & 255]
        ^ t2[c>>8 & 255] ^ t3[d & 255] ^
      key[kIndex & 63];
   b2 = t0[b>>>24 & 255] ^ t1[c>>16 & 255]
        ^ t2[d>>8 & 255] ^ t3[a & 255] ^
      key[(kIndex + 1) & 63];
   c2 = t0[c>>>24 & 255] ^ t1[d>>16 & 255]
        ^ t2[a>>8 & 255] ^ t3[b & 255] ^
      key[(kIndex + 2) & 63];
   d  = t0[d>>>24 & 255] ^ t1[a>>16 & 255]
        ^ t2[b>>8 & 255] ^ t3[c & 255] ^
      key[(kIndex + 3) & 63];
   kIndex += 4; a = a2; b = b2; c = c2;
  }

  for (i = 0; i < 4; i++)
  {
   out[(!dir ? i : (3& i)) & 3] =
   sbox[a>>>24 & 255]<<24 ^
   sbox[b>>16  & 255]<<16 ^
   sbox[c>>8   & 255]<<8  ^
   sbox[d      & 255]     ^
   key[kIndex++ & 63];
   a2=a; a=b; b=c; c=d; d=a2;
  }

  return out;
}
```

Figure 4: Main function and final type of AES-256

In all cases, the defensive version was at least as fast as the SJCL one with gains of up to 100%

Final type of AES library object

```
{ASCII:string, STR2WA:(string)[{ASCII:string}]
    => number array[4] array[3], Stables:
   number array[256] array[5] array[2], cbc:(
   number array[>=4] array[>=0], number array
   [>=4], boolean)[{crypt:(number array[4],
   number) => number array[>=4]}] =>
   undefined, crypt:(number array[>=4],
   number)[{Stables:number array[>=256] array
   [>=5] array[>=2], key:number array[>=64]
   array[>=2]}] => number array[4], key:
   number array[64] array[2], setKey:(number
   array[>=8])[{Stables:number array[>=256]
   array[>=5] array[>=2], key:number array
   [>=64] array[>=2]}] => undefined} final
```

on Chrome and 30% on Firefox. The removal of the exception handler for malformed input is the main speedup factor, while on Firefox, the JIT compiler is able to generate more efficient code paths from the defensive code.

Applications We can then use these primitives to build protocols that can run in untrusted JavaScript heaps. Going back to the LastPass bookmarklet example, we can use a simple challenge-handshake authentication protocol based on SHA-256 to protect the bookmarklet's secret. In its simplest form, the bookmarklet can create an iframe in the LastPass origin that will authenticate the bookmarklet using the CHAP protocol and decrypt the page's password on success.

3.5 Related Work

Most existing research on web security rely on the hypothesis that the trusted code is running first. Some do use type systems, for instance for the verification of ADsafety sandboxing properties [8]. Our work show that it is also possible to consider trused code being injected into an untrusted page without leaking any secret to it. However, what we were lacking is a complete operational semantics of JavaScript that would allow us to prove a formal security property. In that regard, there has been significant progress from the F* team at Microsoft Research, which we met in Seattle, using a very simple semantic of the core features of JavaScript called λJS [39].
Using an encoding of the various JavaScript features into that core, they are able to give a formal semantic of ECMA 5 and use the refinement type system of F* to compile ML code into JavaScript and prove a full abstraction theorem for this transformation [40], i.e. that contextual semantic equivalence is preserved by this transformation, which was our initial security goal as well for defensive Javascript. Still, their approach has drawbacks: it relies on a two part process starting from a syntactic conversion of ML into Javascript followed by a complex stage of wrapping the resulting objects in a defensive way. This requires significant code rewrite in a language web developers may not be familiar with and adds significant overhead to the generated code.
Yet it seems that the method of their proof can be used to obtain the security theorem we are missing, more specifically we can try to use their enriched λJS semantics (called JS*) as our target intermediate language.

4 Conclusion and perspectives

In the course of this intership, we have addressed the problem of code running second in an untrusted scope. We have designed and implemented a defensive subset of JavaScript with regard to a malicius execution environment and a program able to check the membership to that subset of any input JavaScript program. We have also implemented a lightweight cryptographic library allowing us to have authenticated and encrypted communication across an unsafe host.
Nevertheless, we are missing a formal soundness proof of our defensiveness claim. Additionally, we have to acknowledge that our current tool, while sufficient for our goals, has a limited expressive power due to its basic type system and

in particular refinement types associated with an SMT solver would have allowed us to automatically check almost all bound conditions without resorting to dynamic checks. Lastly, we are aware that the defensive subset we presented is far from being maximal. In particular, it is possible to have defensive constructors with one level of prototype inheritence and using computational security, to allocate arbitratily large new arrays and objects. In any case, this line of research will be prolonged into a PhD thesis.

References

[1] Same origin policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[2] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from google chrome. *Commun. ACM*, 52(8):45–49, August 2009.

[3] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. Mashupos: operating system abstractions for client mashups. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 16:1–16:7, Berkeley, CA, USA, 2007. USENIX Association.

[4] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[5] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *SIGPLAN Not.*, 44(8):20–31, December 2009.

[6] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. pages 1–16.

[7] Andrei Sabelfeld and Daniel Hedin. Information-flow security for a core of javascript. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, Cambridge, MA, USA, June 2012. IEEE. To appear.

[8] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[9] Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based web. `http://code.google.com/p/google-caja/`.

[10] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, Cambridge, MA, USA, June 2012. IEEE. To appear.

[11] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[12] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *6th USENIX Workshop on Offensive Technologies (WOOT'12)*. Usenix, aug 2012. To appear.

[13] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Proceedings of the 14th international conference on Financial cryptograpy and data security*, FC'10, pages 136–149, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Clipperz. `http://clipperz.com`.

[15] SpiderOak. http://spideroak.com.

[16] Wuala. http://wuala.com.

[17] LastPass. http://lastpass.com.

[18] 1Password. https://agilebits.com.

[19] RoboForm. http://www.roboform.com.

[20] PKCS #5: Password-Based Cryptography Specification, Version 2.0. IETF, 2000.

[21] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Proceedings of the First International Workshop on Information Security*, ISW '97, pages 121–134, London, UK, UK, 1998. Springer-Verlag.

[22] Andrey Belenko and Dmitry Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Co. Ltd., 2012. http://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf.

[23] Steven M. Bellovin. Cryptography and the internet. In *Advances in Cryptology: Proceedings of CRYPTO '98*, August 1998.

[24] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 98–107, New York, NY, USA, 2002. ACM.

[25] CVE-2012-3882: Roboform "Receive Passcard by E-mail" Feature Accepts Tampered Metadata, July 8 2012.

[26] CVE-2012-3883: 1Password Restore Feature Accepts Tampered Metadata, July 8 2012.

[27] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 75–88. ACM, 2008.

[28] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Rogert Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, SRDS '06, pages 189–198, 2006.

[29] CVE-2012-3874: Wuala Status Page Leaks Plaintext Files, July 7 2012.

[30] RFC3986: Uniform Resource Identifier (URI): Generic Syntax. IETF, 2005.

[31] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[32] Parseuri 1.2: Split urls in javascript. http://stevenlevithan.com/demo/parseuri/js/.

[33] CVE-2012-3879: Phishing attack on 1Password Browser Extensions, July 8 2012.

[34] Ben Adida, Adam Barth, and Collin Jackson. Rootkits for JavaScript environments. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, 2009.

[35] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October–December 2008. Special issue IEEE Symposium on Security and Privacy 2006. Electronic version available at http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.

[36] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, pages 125–140, 2010.

[37] Zhengqin Luo and Tamara Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. *Computer Security Foundations Symposium, IEEE*, 0:157–170, 2012.

[38] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *APLAS*, pages 307–325, 2008.

[39] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *In ECOOP, Lecture Notes in Computer Science*, pages 40–52. Springer, 2004.

[40] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, May 2011.