Language-based Defenses against Untrusted Browser Origins

Karthikeyan Bhargavan
INRIA Paris-Rocquencourt

Antoine Delignat-Lavaud
INRIA Paris-Rocquencourt

Sergio Maffeis
Imperial College London

Abstract

We present new attacks and robust countermeasures for security-sensitive components, such as single sign-on APIs and client-side cryptographic libraries, that need to be safely deployed on untrusted web pages. We show how failing to isolate such components leaves them vulnerable to attacks both from the hosting website and other components running on the same page. These attacks are not prevented by browser security mechanisms alone, because they are caused by code interacting within the same origin. To mitigate these attacks, we propose to combine fine-grained component isolation at the JavaScript level with cryptographic mechanisms. We present Defensive JavaScript (DJS), a subset of the language that guarantees the behavior integrity of scripts even when loaded in a hostile environment. We give a sound type system, type inference tool, and build defensive libraries for cryptography and data encodings. We show the effectiveness of our solution by implementing several applications using defensive patterns that fix some of our original attacks. We present a model extraction tool to analyze the security properties of our applications using a cryptographic protocol verifier.

1 Defensive Web Components

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 [23] 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.
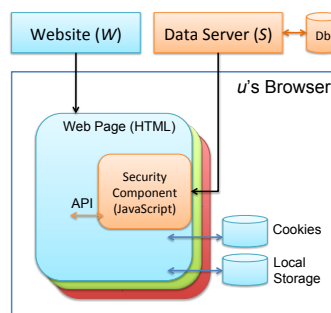

Figure 1: JavaScript Security Component

The typical deployment scenario that concerns us is depicted in Figure 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.

A popular example of this scenario is single sign-on mechanism, such as Login with Facebook (detailed in Section 2). 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 CSRF-Guard, 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 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 [1, 5, 6, 10, 41, 42]). What makes such components so hard to secure?

In Section 2, 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 builtin 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.

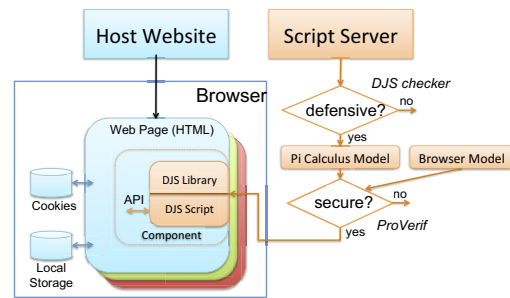**Same Origin Policy (SOP).** Most browser security mechanisms (including new HTML5 APIs, such as

Figure 2: DJS Architecture

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, 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) 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 to the applied pi calculus for verification when combined with models of the browser and DJS library, using the ProVerif protocol analyzer.

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 code. In order to verify authentication and secrecy properties of the defensive components once embedded in the browser, we rely on ProVerif [13], a standard protocol verification tool that has been used extensively to analyze cryptographic mechanisms, with the WebSpi library [6], 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 6, 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 [16] or Caja [40], 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.

Supporting materials for this paper, including code, demos, and a technical report with proofs are available online [11].

2 Attacks on Web Security Components

We survey a series of web security components and investigate their security; Table 1 presents our results. Our survey focuses on three categories of security components that implement the pattern depicted in Figure 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.

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.

| 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 Verisign, SuperGenPass	Password Manager	Bookmarklet, Frames, JavaScript Crypto	Malicious JavaScript URL Parsing Confusion	Bookmarklet Secret, Encryption Key
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 1: Survey: Representative Attacks on Security Components

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 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 [11].

2.1 Login with Facebook
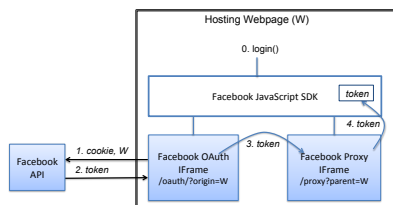
When a website W wants to incorporate single-sign on with Facebook (S) on one of its pages, it can simply include the Facebook JavaScript SDK and call `FB.login()`. Behind the scene, this kicks off a three-party authoriza-

tion protocol called OAuth 2.0 [23], 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 (I_W) as parameter:
https://www.facebook.com/dialog/oauth?client_id=I$_W$
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 [43]. 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) [38], 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 [6].

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.

2.2 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 builtin 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 [1]. 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 [37]. 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.

2.3 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 [5, 6, 10]; 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.

3 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. [16, 26, 39, 40] or require trusted code to run first (e.g. [2]).

The rest of the section informally describes the DJS subset and its security properties; full formal definitions can be found in the technical report [11].

3.1 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 [18] 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 [22].

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:

```
E = (function() {
  var F = function(x) {
      var secret = 42, key = 0xC0C0ACAFE;
      return x===key ? secret : 0 }
  return function G_F(x) { return F(x>>>0) }
})();
```

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

Definition 1 (Encapsulation). A function wrapper E encapsulates F over domain $\mathscr{D}$ 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 $\tilde{v} \in \mathscr{D}$, the heap resulting from calling $G_F(\tilde{v})$ is equivalent to the heap resulting from calling $F(\tilde{v})$.

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 $\mathscr{D}$. 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* [19]: 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.

<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 3: DJS Syntax.

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 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.

3.2 DJS Language

In practice, JavaScript code is considered valid DJS if it is accepted by the automatic conformance checker described in Section 4.1, which in turn is based on the type system of Section 3.3. The type system effectively imposes a restricted grammar on DJS that is given in Figure 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 1 and 2); 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}:

```
1 (function(){
2   var _ = <function>;
3   return function(x){
4     if(typeof x == "string") return _(x)}
5 })();
```

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

```
1 function (<id>,...,<id>){
2   var <id> = <expr>,...,<id> = <expr>;
3   <statements>
4   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 pro-

tect 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 [21]: 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 3.1, we can avoid the second attack by running wrapped DJS code within a `try` block. To avoid the third and fourth

Types and Environments.

$$\langle\tau\rangle ::= \text{number} \mid \text{boolean} \mid \text{string} \mid \text{undefined} \qquad \text{Base types}$$
$$\mid \quad \bar{\tau} \to \tau \qquad\qquad\qquad\qquad\qquad\qquad \text{Function}$$
$$\mid \quad \bar{\tau}[\rho] \to \tau \qquad\qquad \text{Method operating on properties } \rho$$
$$\mid \quad \delta \qquad\qquad\qquad\qquad\qquad \text{Objects and arrays}$$

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

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

$[\sigma^* \text{ and } \sigma \text{ are same thing sometimes}]$

Subtyping.

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

$$\frac{v_1 <: v_2 \quad \bar{\mu}_2 <: \bar{\mu}_1}{\bar{\mu}_1 \to v_1 <: \bar{\mu}_2 \to v_2} \qquad \frac{\rho_2 <: \rho_1 \quad \bar{\mu}_1 \to v_1 <: \bar{\mu}_2 \to v_2}{\bar{\mu}_1[\rho_1] \to v_1 <: \bar{\mu}_2[\rho_2] \to v_2}$$

Figure 4: DJS types, subtyping and environments.

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 5 and 6, 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.

3.3 Type System

DJS types and their subtyping relation are defined in Figure 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 [14, 15, 33]. 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

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

$$\text{Arr}\,\frac{\Gamma \vdash e_i : \tau \quad i \in [1..n]}{\Gamma \vdash [e_1, \ldots, e_n] : [\tau]_n^*} \qquad \text{StrD}\,\frac{\Gamma \vdash x : \text{string} \quad \Gamma \vdash y : \text{number}}{\Gamma \vdash ((y \gg = 0) < x.\texttt{length}?x[y] : @string) : \text{string}} \qquad \text{ArrD}\,\frac{\Gamma \vdash x : [\tau]_n \quad \Gamma \vdash e : \text{number} \quad n > 0}{\Gamma \vdash x[(e \gg 0)\%x.\texttt{length}] : \tau}$$

$$\text{Scope}\,\frac{\Phi(x) = \tau}{\Gamma, [\Phi]_\kappa \vdash x : \tau} \qquad \text{RecScope}\,\frac{x \notin dom(\Phi) \quad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_\mathsf{s} \vdash x : \tau} \qquad \text{FunDef}\,\frac{\begin{array}{c}\Gamma, [\tilde{x} : \tilde{\alpha}, (y_i : \mu_i)_{i<j}]_\mathsf{s} \vdash e_j : \mu_j \quad j \in [1..m] \\ \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_\mathsf{s} \vdash s : \text{undefined} \quad \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_\mathsf{s} \vdash r : \tau\end{array}}{\Gamma \vdash \text{function } (\tilde{x})\{\text{var } y_1 = e_1, \ldots, y_m = e_m; s; \text{return } r\} : \tilde{\alpha} \to \tau}$$

$$\text{Assign}\,\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \tau} \qquad \text{With}\,\frac{\Gamma \vdash e : \{\tilde{x} : \tilde{\tau}\} \quad \Gamma, [\tilde{x} : \tilde{\tau}]_\mathsf{o} \vdash s : \text{undefined}}{\Gamma \vdash \text{with}(e)s : \text{undefined}} \qquad \text{MetDef}\,\frac{\Gamma \vdash \text{function } (this, \tilde{x})\{s\} : (\rho, \tilde{\alpha}) \to \tau}{\Gamma \vdash \text{function } (\tilde{x})\{s\} : \tilde{\alpha}[\rho] \to \tau}$$

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

Figure 5: Selected typing rules.

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; here we only discuss a few representative examples, reported in Figure 5; the other typing rules are detailed in the full version [11]. 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 type of its parent object l.

Formal Guarantees. The DJS type system enjoys both *type soundness* (types are preserved by computation) and *progress* (typed programs terminate with a final value and do not raise exceptions). A consequence of type soundness is that well-typed programs are defensive. All formal definitions and proofs leading to Theorem 1 can be found in the technical report [11].

Theorem 1 (Defensiveness). *If $\emptyset \vdash F$:string $\to$ string then the DJS wrapper E_{DJS} encapsulates F over strings and preserves its independence.*

Another consequence of type soundness is that the execution of well-typed programs does not affect attacker memory [11]. As a consequence, execution of DJS programs is invisible to the attacker.

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.

4 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.

```
# ./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 6: Screenshot of the DJS tool: first a type-checking error, then a (cut off) ProVerif translation.

4.1 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 6 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: `{a:number,b:number}`, `{a:number}`, `{b:number}` or `{}`. 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 $\tau = $ 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 $X \subseteq \{x_i : T_i\}_{i \in [1..n]}$.

Our tool uses type inference as a heuristic, and relies on the soundness of the type checking rules of Section 3.3 for its correctness. Our inference and unification algorithms are standard. We refer interested readers to our implementation for additional details.
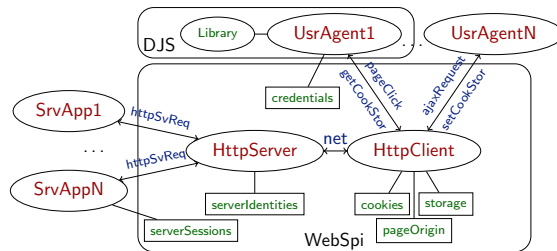

Figure 7: WebSpi model and DJS components

4.2 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 consist of several scripts exchanging encrypted messages with each other and with other frames and websites. 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.

In prior work, the WebSpi library of the ProVerif tool has been used to analyze the security of web applications [5, 6]. The main processes, channels and data tables of WebSpi are represented on Figure 7. `UsrAgent` processes model the behavior of JavaScript running on a page, while the other processes handle communications and processing of server requests.

The advantage of this methodology is that an application can be automatically verified against entire classes of web attackers. ProVerif can handle an unbounded number of sessions, but may fail to terminate. If it verifies a model, it can serve to increase confidence in the security application. The disadvantage is that to model a JavaScript component in WebSpi, a programmer normally has to write an applied pi calculus process for each script by hand.

We developed a model extraction tool that automatically generates user agent process models of components written in the subset of DJS without loops, using a process and data constructor library for cryptographic operations and serialization (matching our implemented DJS libraries introduced in the next section).

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.

We do not fully detail our translation from DJS to the

applied pi calculus here for lack of space; it follows Milner's famous "functions as processes" encoding of the lambda calculus into the pi calculus [30]. Similar translations to ours have previously been defined (and proved sound) for F# [12] and Java [4]. Our translation only works for well-typed DJS programs that use our DJS libraries; it does not apply to arbitrary JavaScript.

DJS programs may prefix a function name by `_lib` to indicate that the code of certain functions should not be translated to applied pi and they must instead be treated as trusted primitives. A typical example is cryptographic functions, which get translated to symbolic functions.

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 describe complex components we verified in Section 6.

5 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.

5.1 Defensive JavaScript Crypto Library

Our starting points for DJCL are two widely used JavaScript libraries for cryptography: SJCL [37] (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).

5.2 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. Deserializing 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 [11]) 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) [25]. Our library interoperates with other server-side implementations of JOSE

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: Evaluation of DJS codebase

(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.)

6   Applications

We revisit the password manager bookmarklet, single sign-on script, and encrypted storage website examples from Section 2 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 summarizes our codebase and verification results.

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 [1]; indeed, we found new variations of these attacks (Section 2) 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.

6.2   Script-level Token Access Control

The Facebook login component discussed in Section 2 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.
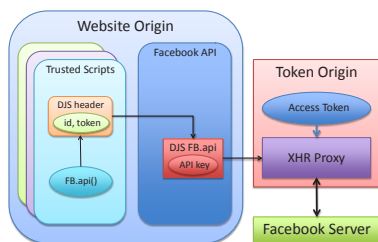
The token origin then provides a proxy frame to the main website (e.g. `*.yahoo.com`) that only allows authorized scripts to use the token. The frame listens for requests signed with JWT using an API key; if the signature is valid, it will inject the access token into the request and forward it to the 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:

```
1 with({sign_request: (function(){
2   var djcl = {/*...*/};
3   var id = "me.js", tok = "1f3c...";
4   var _ = function(s){
5     return s == "/me" /* || s== "..." */ ?
6     djcl.jwt.create(
7       djcl.djson.stringify({jti: id, req: s}), tok
8     ) : "" };
9   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.

6.3 An API for Client-side Encryption

In Section 2 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.

7   Related Work

Attacks similar to the ones we describe in Section 2 have been reported before in the context of password manager bookmarklets [1], frame busting defenses [35], single sign-on protocols [6, 36, 41], payment processing components [42], smartphone password managers [9], and encrypted cloud storage [5, 10]. These works provide further evidence for the need for defensive programming techniques and automated analysis for web applications.

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 [2, 7, 8, 27, 44]. 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 variety of JavaScript subsets attempt to protect trusted web pages from untrusted [20, 26, 28, 29, 31, 32, 34, 39]. 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 scripts do not need such privileges.

[21] 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.

Dynamic information flow analyses for various subsets of JavaScript [3, 17, 24] enforce a security property called noninterference. Our static type system enforces defensiveness and we analyze security by model extraction. Relating defensiveness to noninterference remains future work; we conjecture that DJS may be more suitable than JavaScript to static information flow analysis.

8 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, this paper presents 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.

References

[1] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript environments. In *WOOT*, 2009.

[2] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *USENIX Security*, 2012.

[3] T. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

[4] M. Avalle, A. Pironti, D. Pozza, and R. Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2:34–48, 2011.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *POST*, 2013.

[6] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF*, pages 247–262, 2012.

[7] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript mashup communication. In *W2SP*, 2009.

[8] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *USENIX Security*, 2008.

[9] A. Belenko and D. Sklyarov. "Secure password managers" and "Military-grade encryption" on smartphones: Oh, really? Technical report, Elcomsoft Ltd., 2012.

[10] K. Bhargavan and A. Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *WOOT*, 2012.

[11] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive JavaScript website with testbed, technical report and supporting materials. `http://www.defensivejs.com`, 2013.

[12] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152, 2006.

[13] B. Blanchet and B. Smyth. *ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. `http://www.proverif.inria.fr/manual.pdf`.

[14] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.

[15] L. Cardelli. Extensible records in a pure calculus of subtyping. In *In Theoretical Aspects of Object-Oriented Programming*, pages 373–425. MIT Press, 1994.

[16] D. Crockford. ADsafe: Making JavaScript safe for advertising. `http://www.adsafe.org/`, 2008.

[17] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, pages 748–759, 2012.

[18] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[19] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *CCS*, pages 161–174. ACM, 2008.

[20] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *BDSS*, 2010.

[21] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL'13*, 2013.

[22] P. Haack. JSON hijacking. `http://hhacked.com/2009/06/25/json-hijacking.aspx`, 2009.

[23] D. Hardt. The OAuth 2.0 authorization framework. IETF RFC 6749, 2012.

[24] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF*, pages 3–18, 2012.

[25] IETF. *JavaScript Object Signing and Encryption (JOSE)*, 2012. `http://tools.ietf.org/wg/jose/`.

[26] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS'09*, 2009.

[27] L. Meyerovich, A. Porter Felt, and M. Miller. Object views: Fine-grained sharing in browsers. In *WWW*, 2010.

[28] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, 2010.

[29] J. Mickens and M. Finifter. Jigsaw: Efficient, low-effort mashup isolation. In *USENIX Web Application Development*, 2012.

[30] R. Milner. Functions as processes. In *Automata, Languages and Programming*, volume 443, pages 167–180. 1990.

[31] P. Phung, D. Sands, and D. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.

[32] J. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[33] F. Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.

[34] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.

[35] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP'10*, 2010.

[36] J. Somorovsky, A. Mayer, A. Worth, J. Schwenk, M. Kampmann, and M. Jensen. On breaking SAML: Be whoever you want to be. In *WOOT*, 2012.

[37] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, pages 373–381, 2009.

[38] B. Sterne and A. Barth. Content Security Policy 1.0. W3C Candidate Recommendation, 2012.

[39] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE S&P*, 2011.

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

[41] R. Wang, S. Chen, and X. 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 S&P*, pages 365–379. IEEE Computer Society, 2012.

[42] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *IEEE S&P*, pages 465–480, 2011.

[43] M. Zalewski. *The Tangled Web*. No Starch Press, November 2011.

[44] L. Zhengqin and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. 2012.