Embedding of Security Components in Untrusted Third-Party Websites

Antoine Delignat-Lavaud, Karthikeyan Bhargavan, Sergio Maffeis

Embedding of Security Components in Untrusted Third-Party Websites

Antoine Delignat-Lavaud, Karthikeyan Bhargavan, Sergio Maffeis

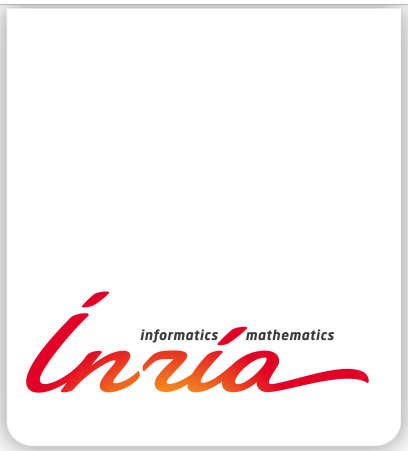
RESEARCH REPORT

N° 8285

April 2013

Project-Team PROSECCO

ISSN 0249-6399 ISRN INRIA/RR--8285--FR+ENG



Embedding of Security Components in Untrusted Third-Party Websites

Antoine Delignat-Lavaud, Karthikeyan Bhargavan, Sergio Maffeis

Project-Team PROSECCO

Research Report n° 8285 — April 2013 — 32 pages

Abstract: Security-sensitive components, such as single sign-on APIs, need to be safely deployed on untrusted webpages. We present several new attacks on security components used in popular web applications that demonstrate how failing to isolate such components leaves them vulnerable to attacks both from the hosting website and other components loaded 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 behavioral integrity of trusted scripts loaded in an untrusted page. 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 isolation patterns that fix some of our original attacks. We use a translation of a fragment of DJS to to applied pi-calculus to verify concrete security policies of critical components against various classes of web attackers.

Key-words: JavaScript, security, cross-site scripting, cross-site request forgery, browser cryptography, single sign-on, encrypted cloud storage

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Chargement d'interfaces de scurit dans des sites web tiers

Résumé : Certaines librairies critiques pour la scurit, par exemple pour l'authentification unique (single sign-on), ncssitent d'tre charges dans des sites tiers non srs. Nous montrons comment le manque d'isolation entre ces librairies et leur page hte les rendent vulnrables aux scripts tiers qui partagent le mme environnement, y compris pour des services trs largements utiliss. Les mcanismes de scurit des navigateurs sont impuissants face ces attaques car elles interviennent en dessous de la granularit des politiques de scurit, fixe par origine (protocole, nom de domaine et numro de port). Afin de mitiger ces attaques, nous proposons de combiner une isolation fine de ces librairies au niveau du langage avec des protections cryptographiques. cette fin, nous introduisons DJS, un fragment de JavaScript qui protge l'intgrit de l'excution d'un script dans un environnement JavaScript hostile. Nous nous appuyons sur un systme de types infrables et sur un ensemble de librairies (elles-mme bien types) pour implmenter des solutions gnriques aux attaques que nous avons dcouvertes. Nous vrifions la validit de ces schmas l'aide d'une traduction d'un sous-ensemble de DJS vers le pi-calcul appliqu contre diffrent types d'attaques.

Mots-clés : JavaScript, security, cross-site scripting, cross-site request forgery, browser cryptography, single sign-on, encrypted cloud storage

1 Defensive Security on Untrusted Websites

Web users increasingly store sensitive personal 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 disadvantage is that the concentration of sensitive data on servers makes them tempting targets for cybercriminals, 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 and cryptographic mechanisms. These mechanisms are typically embedded as JavaScript components that run on partially-trusted web pages and protect user data from unauthorized parties, until the user explicitly gives her permission by entering a password or clicking on a button.

Popular examples of such components include single sign-on mechanisms such as Login with Facebook, social sharing widgets such as the ubiquitous Like button, payment processing APIs such as Google Checkout, password manager bookmarklets such as Lastpass, and client-side encryption libraries used by cloud storage services such as Wuala.

We investigate the security of such components when they are loaded on untrusted websites alongside other third-party components, including frameworks such as jQuery or libraries such as Google Analytics. Can the component be protected from interference or attacks from other code running on the same webpage?

Same Origin Policy. Most browser-based security mechanisms 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`). The Same Origin Policy of the browser isolates JavaScript global environments associated with different origins, while allowing code from different global environments associated to the same origin to interact, for example by reading each other's global variables or calling each other's functions. Scripts included into a page using the `<script>` tag share the global environment of that page regardless of the URL from which they were sourced. Frames sourced from the same origin can access each other's documents. Newer HTML5 APIs, such as postMessage, localStorage, and WebCrypto, all enforce access control based on origin.

Hence, security components can rely on the browser to protect their secrets and functionality from malicious components on other origins. Conversely, if a malicious script is loaded in any page on an origin, it may access all capabilities available to that origin.

Attacks on Security Components. In Section 3, we survey the state of the art in two classes of security components: OAuth 2.0 single sign-on mechanisms (e.g. "Login with Facebook") and client-side encryption libraries used for cloud storage applications. These components are specifically designed to be deployed on partially trusted websites. So, they isolate their code using a variety of techniques, such as frames, browser extensions, and Java applets. Secure communication within a component or with the hosting page then relies on some combination of cryptography with HTML5 messaging and storage APIs.

Despite their use of such sophisticated mechanisms, we still found attacks against many of these components that completely break their stated security goals. Table 1 summarizes some of these attacks. These attacks broadly fall in three categories:

- *Origin spoofing*: a malicious website fools a security component into thinking that it has been loaded into a different, trusted origin.
- *Rootkits*: a malicious script that runs before a security component tampers with the JavaScript environment to affects the component's functionality and steals its secrets.
- *Cross-Site Scripting (XSS)*: an XSS vulnerability on any page in a trusted ori-

gin enables an attacker to inject a malicious script that can interact with a security component on that origin and steal its secrets.

These kinds of attacks are well known, but our study shows that they are difficult to avoid even for security-conscious developers.

Rather than suggest ad hoc point-fixes to specific vulnerabilities, we advocate a new *defensive* programming style with strong formal guarantees against entire classes of attacks. We show how developers may write defensive components in JavaScript that combine cryptography and browser security mechanisms, but are still safe to deploy in untrusted environments.

Programming Securely in JavaScript. The security components we consider in this paper are written primarily in JavaScript and use a number of HTML5 APIs. Before proceeding, it is perhaps worth questioning whether JavaScript (or HTML5) is even a suitable programming environment for security.

To take an instructive example, JavaScript cryptography has proved to be quite controversial. On the one hand, a number of web applications use JavaScript cryptographic libraries like the Stanford JavaScript Crypto Library (SJCL) [31]. On the other hand, security experts criticize the use of such libraries in comparison with both traditional desktop applications and server-based cryptography (see for example [29]). The gist of these argument is that: (1) a crypto library downloaded from a server does not protect the user against server-side tampering (due to hacking) or from client-side tampering (due to XSS); (2) JavaScript is too flexible for cryptography, since it allows basic language constructs to be redefined; (3) JavaScript lacks core functionality like a cryptographically strong random number generator; (4) JavaScript runtimes, being new and untested, may be more vulnerable to side-channel attacks.

Do these criticisms mean that cryptographic libraries like SJCL and components that use them are doomed to be insecure? In Sections 6 and 7, we show that it is possible to avoid some of these pitfalls as long as one is willing to program both the cryptographic library and its applications in a restricted subset, and as long as one can rely upon some trusted server for some tasks.

Notably, in our cryptographic applications, we address the four concerns above as follows: (1) we download security components from a trusted third-party script server; the same origin policy of the browser then protects our scripts from client-side tampering; (2) we write our crypto library in a defensive subset of JavaScript that ensures independence from the untrusted environment; (3) we rely on the script server to provide us with entropy for generating keys; (4) we never expose any long-term keys to the browser, hence any side-channel attacks may only expose short-term session keys. In short, we forbid random number generation and long-term key storage on the browser, we rely on a third-party server for code distribution and key management, and program defensively in a subset of JavaScript.

Towards Defensive JavaScript Components. Requiring that all scripts on an origin be trusted is too demanding. Instead, we advocate that security components be designed in a way that they resist attacks. Rather than rely on origin-based security mechanisms, we investigate language-based isolation for such security components. This leads us to identify the *defensive JavaScript problem*:

Define a defensive *subset of JavaScript to write stateful functions whose behavior cannot be influenced (other than by varying their input parameters) by untrusted code running in the same JavaScript environment, whether the code is run before or after installing such functions. Moreover, 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 [12] or Caja [11], which focus on protecting a trusted page by sandboxing untrusted components. In particular, those subsets rely on the assumption that the initial JavaScript environment is trusted (no rootkit is installed) and that the

untrusted code can be restricted to ensure that it cannot escape its sandbox. In our case, defensive code must run securely in a potentially rootkitted JavaScript engine, and the untrusted code is arbitrary.

Our Approach and Contributions. We present DJS, a defensive subset of JavaScript for programming security components that can run safely on untrusted web pages. To our knowledge, DJS is the first language-based isolation mechanism that does not restrict untrusted JavaScript, or even the surrounding HTML5 environment, and that does not need to be loaded first in the browser.

We use DJS to program defensive libraries for cryptography and JSON serialization. Building on DJS programs, defensive cryptography, and browser-based security mechanisms such as iframes, we show how to build defensive components with strong security guarantees, even in malicious environments. We use these components to fix some of the web application attacks we found.

Our main contributions are:

- We identify common concerns for applications that embed secure components in arbitrary 3rd party websites, and identify new attacks specific to these applications.
- We propose DJS as a linguistic solution to program defensive security components.
- We give an automated tool to verify if a piece of JavaScript code is valid DJS.
- We define DJCL, a defensive crypto library with encoding and decoding utilities that can be safely used in untrusted JavaScript environments.
- 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.

Demonstrations of our type inference program, defensive scripts and an up-to-date copy of this technical report are available anonymously at `http://www.defensivejs.com`.

Outline. Section 2 briefly discusses related work. Section 3 studies common security components and presents new attacks on popular web applications that motivate defensive JavaScript. Section 4 introduces DJS. Section 5 states our main security theorem about DJS programs. Section 6 describes our defensive libraries and Section 7 describes several defensive applications. Section 9 concludes.

2 Related Work

Attacks on JavaScript Security Components. Our threat model, of untrusted scripts that tamper with a web page before a security component is run, is sometimes called a *JavaScript rootkit*. It has been explored both in the context of bookmarklets that try to authenticate the origin in which they are loaded [2], and in the context of frame busting defences, by which a web page tries to detect if it is loaded inside an `iframe` element [28]. In this paper, we generalize the threat model to address defensive programming in a wider variety of applications.

Many attacks on single sign-on components in websites have been discovered in recent years, including XML-rewriting attacks on SAML [30], token redirection and CSRF attacks on OAuth 2.0 [6], and a variety of logical flaws in OpenID and OAuth implementations [33]. The last two of these list attacks that are similar in flavor to our attacks of Section 3. In particular, [33] reports a Flash-specific attack on the Facebook single sign-on component. Other recent attacks related to the problems considered in this paper include those on payment processing components [34], password manager key derivation [9], and various applications for encrypted cloud storage [10].

We cite these works as further evidence for the need for provably secure defensive programming techniques for web security components. None of the above papers proposes any generic countermeasures beyond point-fixes to specific vulnerabilities. In contrast, we propose DJS for preventing some of these attacks and show how to use it with cryptography to program secure applications.

Category	Example	Isolation Mechanism	Threat	Attack
Single Sign-On Library	Facebook JS SDK	Frames	Token Phishing	Origin Spoofing
Single Sign-On Client	Yahoo (Facebook Login)	Frames	Open Redirector	Token Redirection
Web Browser	Firefox	Frames, CSP	Cross-Origin Access	Redirection URI Theft
Password Manager	1Password	Browser Extension	Phishing	Origin Spoofing
Password Form Filler	LastPass	Bookmarklet, Cryptography	Phishing	Password (Key) Theft
Cloud Storage	SpiderOak	JavaScript Cryptography	CSRF	Shared Folder Theft
Cryptographic Web Application	ConfiChair	Java Cryptography	XSS	Password (Key) Theft

Table 1: Example Attacks on Security Components

Frame-based Isolation. A number of works explore the use of frames and inter-frame communication to isolate untrusted components on a page [8, 7, 20, 22, 3]. These mechanisms typically assume that the parent page is trusted, and a bootstrapping script is run first to set up frame isolation and inter-frame communication. However, in our application scenarios, the hosting page is typically untrusted and the security component must be ready to be loaded into a malicious environment.

A recent work in this category [3] proposes a privilege-separation mechanism for HTML5 applications that isolates all website code except a small trusted script within frames that are given temporary (sandboxed) origins. Accesses to the parent website are performed via the HTML5 postMessage API. To make this work, the website code has to be slightly rewritten to work within a frame, and website interactions such as AJAX calls incur a performance penalty due to cross-frame messaging. In contrast, we propose to only rewrite and isolate security components, leaving untrusted code unchanged. Considering that the vast majority of code on a website is not security-critical, our approach promises better performance, while removing the dependence on running first.

Secure JavaScript Subsets. A variety of JavaScript subsets attempt to restrict access from untrusted scripts to sensitive APIs loaded on trusted web pages [15, 24, 23, 26, 27, 21, 25, 32]. Our goal is instead to run trusted components within untrusted web pages, hence our security goals are stronger, and our language restrictions are different. In practice, many of these solutions 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. For example, [32] propose a subset called Secure ECMAScript in which all untrusted code must be written. Since this subset forbids any modification of language prototypes it is incompatible with popular JavaScript libraries such as Prototype and MooTools. This language restriction is imposed by a bootstrapper that freezes all the language prototypes and hides dangerous APIs. In our setting, the attacker runs first, and such defences are not available. Moreover, we only want to restrict the security-sensitive website code.

Trusted Wrappers for JavaScript. Recent work [16] explores whether trusted code can be protected from an untrusted JavaScript environment. All trusted code is written in the F* functional language, and then compiled to JavaScript. Their main theorem states that two F* programs are behaviorally equivalent in F* if and only if their translations are behaviorally equivalent in JavaScript. That is, the JavaScript environment cannot break the secrecy and authenticity invariants However, we found that malicious code in the HTML document can quite effectively break the security invariants of their JavaScript code. We discovered several counterexamples to their defensive wrappers and security applications. In particular, F* programs that hide sensitive DOM functions do not hide them anymore when translated to JavaScript, and secrets stored in F* programs may be leaked by their JavaScript translations. We have reported our counterexamples to the authors of [16], leading to fixes in their defensive wrappers and ongoing discussions on extending their results to our attacker model.

Secure Information Flow for JavaScript. Several recent works [19, 13, 5] propose information flow analyses for various subsets of JavaScript

that aim to enforce a form of noninterference; put simply, high-security data is kept isolated from low-security data. These analyses are typically implemented as dynamic checks at runtime, since static analysis is infeasible for general JavaScript programs. In contrast, we present a static analysis that identifies a subset of JavaScript for which a different property called defensiveness holds. Defensiveness does not guarantee security; defensive programs may still leak secrets or accept tainted data. However, it does guarantee a form of functional integrity that we call independence. Relating defensiveness formally to noninterference remains future work, but we conjecture that programs written in our defensive subset of JavaScript may lend themselves more easily to information flow analysis.

3 Security Components: Examples and Attacks

Table **??** summarizes attacks that we found on a number of commercial security components. Each row in the table is corresponds to one category of components and is representative; we typically found similar attacks on a number of other components in the same category.

In this Section, we study in detail the implementations of two security components that try to protect user data from other components on the same (untrusted) web page. In each case, we show how the component can be attacked by a malicious script on the hosting page, and argue that their security should be independent of their environment. Our attacks bypass existing sophisticated defenses in these applications amd demostrate the need for verified defensive programming techniques for JavaScript.

3.1 OAuth 2.0: Single Sign-On and API Authorization

Websites such as Facebook, Google, and Live, protect API access to user data through the OAuth 2.0 [18] authorization protocol. To access user data from any of these websites, a web application must first obtain an *access token* from an authorization server. The token is issued only if the user explicitly gives her permission.

A typical use for such tokens is to authenticate a user, by retrieving her email address. This is the basis of single sign-on solutions provided by various social networks (e.g. Login with Facebook). More generally, the token may be used to retrieve user data for personalization, or to post a comment on a user's social profile (e.g. Like buttons).

All major authorization providers offer JavaScript SDKs to allow web applications to easily request and use access tokens through a simple client-side API. We consider the design of the Facebook JavaScript SDK. In particular, the SDK exports the FB.login() method that contacts Facebook to verify that the current user is logged in to Facebook, checks that the user has authorized the current website, and retrieves a fresh access token. The page can then access the token directly by calling FB.getAccessToken(), or attach it automatically to API requests using FB.api().

The goal of the login component of the SDK is to ensure that only authorized applications are given a token to access the user data. In its strongest server-side configuration, the OAuth protocol requires each application to authenticate its token request with its identifier and secret key (obtained during registration), and requires that the authorization server only returns the token to a preregistered application URI.

When returning tokens to JavaScript clients, however, authorization servers are more permissive. They do not require application authentication, because storing or using long term secret keys in JavaScript is considered dangerous. They require only that the destination web page be on the same origin as the application's registered URI, because browsers do not offer a reliable way to distinguish one page on an origin from another. If a user authorizes a website to access her Facebook profile, she is authorizing every script on every page that can be loaded in that website's origin. If any of them is malicious, it will be able to obtain the access token by calling FB.getAccessToken().

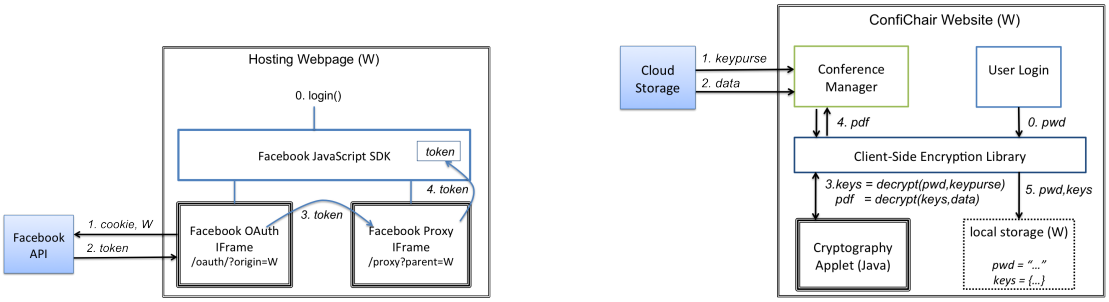


Figure 1: Example security components: (a) Facebook JavaScript SDK single sign-on, (b) ConfiChair website document decryption

We found that even this limited notion of security is not satisfied by common JavaScript login components. Figure 1(a) shows how the Facebook SDK component obtains a token for the current user and website on a browser. When a script on the website W calls FB.login(), two iframes are created. The first *OAuth* iframe is sourced from Facebook's authorization server (`https://www.facebook.com/dialog/oauth?client_id=<W's id>`). This website authenticates the user (with a cookie) and after checking that the user has previously authorized W, it redirects the iframe to a new URL with the access token as a fragment identifier (`https://static.ak.facebook.com/connect/xd_arbiter.php#token=<accesstoken>`). The JavaScript loaded from this page retrieves the access token from the URL and sends it to the second iframe.

The second *Proxy* iframe is sourced from the same webpage that the first frame is redirected to, but with a fragment identifier indicating the origin of the host page (`https://static.ak.facebook.com/connect/xd_arbiter.php#origin=W`). 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 proxyMessage function on the Proxy iframe to deliver the access token. This function then forwards the token to the waiting FB.login callback function on the hosting webpage W, by sending a postMessage event to the parent frame with target origin W.

This is the flow on a trusted website W. To protect against a malicious website M that loads the SDK and tries to steal a user's access tokens by pretending to be W, the login method relies on the following security mechanisms:

1. Both iframes are sourced from origins distinct from the current webpage, so scripts on M cannot look inside the OAuth and Proxy iframes, except to read or write their source URIs.

2. Scripts on M cannot directly access Facebook because the browser and the web server will prevent such cross-origin accesses.

3. The redirection of the OAuth iframe is transparent to the page; so M cannot read the redirection URI and hence the access token in its fragment identifier.

4. When the OAuth iframe calls proxyMessage on the Proxy iframe, the browser would prevent this call unless both frames were on the same origin. Hence, the OAuth frame cannot accidentally leak its token to a different frame or window on M.

5. When the Proxy iframe uses postMessage to send the access token to the parent, it sets the target origin to W; and the browser will refuse to deliver this message to a different origin M.

All these five mechanisms are variations of the so-called browser Same Origin Policy. Essentially, if M and W are different origins, their pages, scripts, frames, and messages are protected from each other.

Origin Spoofing Attacks on the Facebook SDK. We found three ways of confusing the Proxy iframe into releasing its access token to an unauthorized website.

1. The first attack relies on the fact that 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. We wrote a malicious website M that mimics the Facebook login component to open two iframes. It gives the OAuth iframe the application id and origin for W and the Proxy iframe the origin for M. The OAuth iframe duly gets the token for W and passes it to the Proxy iframe that forwards the token to M.

As a result, if a user clicks on a malicious website M, that website can steal the access tokens for any other website that the user has authorized. In practice, our proof-of-concept website can steal authorization codes and login tokens for major websites that enable Facebook login, such as Yahoo, Pinterest, and Live, and also security sensitive services for tax preparation (H&R Block), online health (HealthVault), and banking (Movenbank). Moreover, M can obtain any visiting users' Facebook profile information and write to their walls. This last capability enables M to propagate itself like a worm through the social network.

After our report, Facebook quickly fixed their website to prevent the attack using already existing code for comparing the origins provided to the two frames. However, we found two other ways to bypass this origin comparison, which we subsequently reported and helped fix.

2. Our second attack relies on a weakness in the origin parsing and comparison code of the `xd_arbiter.php` page. In order to enable subdomain x.y.W of W to obtain W's access token, the origin comparison code first extracts the domain from both origins and verifies that the domain provided to the OAuth frame D is a suffix of the domain provided to the Proxy frame D'. More precisely it checks that "."+D is a suffix of "."+D'. However, the domain parsing regular expression is broken, and given an origin `https://M?origin=.W`, it returns `M?origin=.W`. Consequently, if the malicious website gives the Proxy iframe this origin, the origin comparison for W succeeds and the access token for W is leaked to M. (The URL `https://M?origin=.W` is not strictly an origin but is accepted by the postMessage function in all browsers we tested.)

3. Our third attack relies on parameter parsing confusion. If M gives the OAuth frame two origin parameters, the PHP code for the authorization server takes the second parameter as the origin. However, the JavaScript code in the xd_arbiter takes the first parameter as the origin. Consequently, M can get the authorization server to return a token for W but then get the JavaScript origin comparison and authentication to be performed for M.

Breaking Redirection Transparency on Firefox. Our second class of attacks demonstrates how redirection URIs such as the ones used to deliver Facebook access tokens may be inadvertently leaked by a browser.

1. We found that a recent version of the Firefox browser failed to isolate iframe locations. A malicious website M could start an iframe sourced from M, store a pointer to its document.location object, and then redirect the frame to (say) the Facebook OAuth endpoint to obtain a token for W. When the server redirects the frame back to the Proxy endpoint with the access token in its fragment URI, this URI should not be accessible to the parent page M. However, the stored pointer to the frame's location broke this isolation and allowed M to steal W' access token.

2. Our second attack was on Firefox's implementation of Content Security Policy (CSP), a recent proposal for increasing website security against XSS attacks. A notable feature of CSP is that a website can ask the browser to report any access to unauthorized URIs back to the website. Suppose, for example, that the website M asks the browser to block all access to `static.ak.facebook.com` from its pages and report violations of this policy. If the website starts the OAuth iframe pretending to be W, the Facebook OAuth server will issue an access

token for the user and redirect to the blocked URL with a fragment identifier containing the access token. Firefox would then report this violation to M by sending it the full redirection URL, including the access token for W.

Both these attacks enable a malicious website M to obtain the access token for any other authorized website W by relying on bugs in Firefox. We reported these bugs, and they are now fixed.

Inadequacy of Origin-based Authorization. With the fixes above, Facebook access tokens are restricted to the origins for which they are issued. That is still not good enough.

On websites that host user content, such as Dropbox, some parts of the origin may be under the control of a malicious user. For example, we found that user pages hosted under `dl.dropbox.com` were able to obtain access tokens issued for Dropbox (which used the origin `dropbox.com`).

HTTP redirectors on a website may inadvertently leak access tokens. We found an open redirector on the electronic voting website Helios [1] that enabled a malicious website to obtain the access token for any Helios user (and hence log in and vote in her name). Other websites that incorporate redirection into their functionality, such as Yahoo Search and Bitly also enabled malicious websites to steal tokens that were meant for Yahoo and Bitly's login pages.

All these attacks were reported and subsequently fixed by restricting access tokens to subdomains of these websites. However, the problem of authenticating token requests and preventing tokens from leaking in referrer headers and to malicious scripts remains unresolved.

New protocol extensions to OAuth 2.0 such as OpenID Connect attempt to address these problems by introducing cryptography, in the form of signed and encrypted JSON web tokens, to protect tokens from untrusted web pages. We believe this is a promising development, but as we demonstrate in the following section, getting cryptography right comes with its own challenges.

3.2 Client-side Encryption for Host-Proof Web Applications

In response to increasingly effective server-side attacks on databases and websites, some web applications store all user data encrypted on notionally untrusted cloud servers, using keys that are known only to the user, not even to the web application. All plaintext data accesses are performed in the browser, after downloading and decrypting ciphertext from the cloud.

This architecture has been adopted by cloud storage services such as SpiderOak and Wuala, password managers such as LastPass and PassPack, and privacy conscious applications such as Helios, for electronic voting, CryptoCat, for encrypted chats, and ConfiChair, for conference management.

The host-proof design provides an effective defence against offline attacks where a hacker has stolen a server-side database, and cannot decrypt it easily without each user's keys. It is less effective against active adversaries who may tamper with the cryptographic components as they are being delivered to the browser. To address this concern, some applications use signed Java applets, others use JavaScript cryptography within browser extensions, whereas others use reputed JavaScript cryptographic libraries such as SJCL [31].

We consider the ConfiChair conference management system [4] as an illustrative example of incorporating client-side encryption for cloud storage as part of a wider web application. ConfiChair aims to provide its users with formal privacy guarantees that are much stronger than existing systems like EasyChair. A conference chair logs into ConfiChair and starts a new conference and invites a set of users to serve on the program committee. The conference has a public-private keypair (where the private key is known only to the chair) and a shared reviewing key (known to the program committee.). Authors log into ConfiChair and submit papers by encrypting each paper under a fresh symmetric key, which is encrypted under the conference public key and uploaded along with the paper. After submissions are closed, the chair reindexes all the papers (using a mixnet)

and sends them, re-encrypted under the reviewing key, to their allocated reviewers. Reviews are similarly encrypted and sent back to the chair, who finally reveals all review to the corresponding authors. All encrypted papers and encrypted reviews are stored on an untrusted server in the cloud. Each ConfiChair user has a keypurse consisting of all her keys for different roles at various conferences; this keypurse is encrypted under a key derived from her password and also stored in the cloud.

Figure 1(b) shows how the ConfiChair website downloads and decrypts a single paper belonging to an author. The user enters her password on a login page and this password is stored in the HTML5 local storage, but not sent to the server. Instead an authenticator derived from the password is sent to the website. Subsequently, the conference manager may download the user's encrypted keypurse and paper from the cloud server. The client-side encryption library uses a Java applet to first decrypt the keypurse using a key derived from the password, and then uses the appropriate key from the keypurse to decrypt the paper. It caches the decrypted keypurse also in the local storage for subsequent paper downloads and uploads. When the user logs out, her password and keys are deleted from the local storage.

The protocol of ConfiChair has been proven to satisfy several security properties under the following assumptions:

- The paper is encrypted under the keypurse, which is in turn encrypted under the password known only to the user, so an attacker who obtains the keypurse or paper from the cloud or the network cannot read it.
- All cryptography is implemented in the signed Java applet, so it is tamper-proof.
- The password and keypurse are stored temporarily in local storage, but by the same origin policy only pages on the ConfiChair website have access to them.
- We assume that all the JavaScript loaded by the ConfiChair website is loaded from a secure server and cannot been tampered with by a malicious adversary.

However, those result do not consider standard web-based attacks on the ConfiChair website.

An XSS Attack on ConfiChair. The main weakness of the design of ConfiChair is that the password and keypurse are available in local storage to any script running on the ConfiChair origin. Even if all the scripts on ConfiChair are trusted, an XSS attack may introduce a new script that may read and steal this data.

We found an XSS attack on the "change-role" functionality of ConfiChair that enables an arbitrary website to inject a script into the conferences page on ConfiChair. This script can then steal the user's password and keys and send them to the malicious website. So, if an author visits the malicious website, it can read all the authors' submissions. However, if the conference chair visits the website, the entire conference can be compromised.

We reported this vulnerability to the ConfiChair authors and we are in the process of discussing fixes. Despite the XSS attack, one may argue that storing passwords and keys in local storage is bad design but cryptographic web applications often have to balance robust key management with ease of use (e.g. how many times to ask the user for a password). For example, we found a JSONP CSRF attack on the encrypted cloud storage service SpiderOak, that enables malicious websites to steal decryption keys for shared files. The root cause of the attack is that SpiderOak maintains two sets of keys, one for private files and one for shared files. The first set is derived from a user's passphrase, but the second is not, and hence has to be stored on the server, leaving it open to theft via CSRF and other attacks.

Using cryptography is not a guarantee of security. One must question whether the cryptographic library itself can be tampered with, how the keys are managed, and what is the authorization mechanism used before releasing plaintext data to a website. Host-proof web applications have a higher bar of security than normal websites since their whole design is meant to free the user from trusting the server.

Implementing host-proof application correctly demands robustness against common web attacks such as XSS and CSRF. In Section 4 we advocate a defensive programming style for such cryptographic components that trusts as little of the surrounding code as possible.

Rootkit Attacks on Password Manager Bookmarklets. Password manager bookmarklets, such as those provided by LastPass, PassPack and SuperGenPass, provide a second factor of authentication to a password manager; when clicked, they inject cryptographic code into a webpage that retrieves or computes the password for the current page (if any) with the aid of a password manager. We found attacks on all of these bookmarklets that allow malicious websites to tamper with their cryptographic functionality and/or steal their encryption keys. We have yet to encounter any truly defensive commercial cryptographic bookmarklets. In Section 7, we propose a secure login bookmarklet that relies on defensive cryptography in JavaScript.

4 Defensive JavaScript

In this Section we define DJS, a defensive subset of JavaScript for writing trusted components that interact with untrusted webpages while preventing undesirable interferences and attacks. Beside the primary goal of defensiveness, other goals underpinning the design of DJS are: membership must be decidable by a simple static analysis backed up by formal correctness guarantees; run-time performance overhead should be minimal; DJS code should be compatible with existing browsers (running either ECMA Script 3 or 5). In order to address these goals, in some cases we have decided to restrict DJS beyond what is strictly necessary for defensiveness. In Section 4.4 we discuss some extensions to DJS that we believe would not impact its soundness. In practice, an arbitrary JavaScript program is considered valid DJS if it is accepted by the automatic conformance checker described in Section 4.3, which in turn is based on the type system of Section 4.2. A pleasant side effect of DJS typing is that it imposes hygienic coding practices similar to those of the popular JSLint tool, encouraging high quality code that is easy to reason about.

4.1 Threats and Countermeasures

We now summarize the main threats to defensive code and the countermeasures adopted in the design of DJS.

Dependence on non-defensive state. *Threats:* The result returned by a defensive function must depend only on its (defensive) state and on the invocation parameters, so that it can be relied on even when the function is called from within a compromised JavaScript environment. In JavaScript, unexpected 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 the possibility to install getters and setters on object properties. *Countermeasures:* Defensive code should not access variables or call functions defined outside of the defensive code itself. In particular, defensive code should not use global variables, and should not try to access properties of defensive objects that are inherited via the prototype chain from non-defensive objects (this also covers the case of non-defensive getters and setters).

Source code leakage. *Threats:* Defensive scripts are downloaded as source code and cannot call DOM functions (which are under the control of the attacker), therefore any persistent state must be included implicitly in the script itself. For example, a common usage pattern is to include a session-specific nonce that can be used as a source of randomness, as a session identifier or as a key. Therefore, the source code of a defensive function is considered secret, and should not be available to the 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 inlined script from the DOM, or re-loading a remote script as data using AJAX or Flash. Note that since the secret is session-specific, a malicious server that has the power to ignore the SOP and read script sources from any origin, would not be able to obtain the nonce useful for a specific browser session. *Countermeasures:* To avoid the first attack, defensive code only exports to untrusted code 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. To avoid the second attack, defensive code should either be exception-free or catch all possible exceptions. To avoid the third attack, defensive code is never directly inlined in a page. To avoid the fourth attack, defensive scripts are only served from dedicated domains that do not allow cross-domain resource sharing.

Defensive pointer leakage. *Threats:* Since defensive functions can maintain runtime state containing sensitive information, untrusted code should not be able to obtain arbitrary pointers to defensive objects. 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.). *Countermeasures:* Both the JavaScript specification and recent releases of Firefox, Chrome, Internet Explorer and Opera prevent deleting or overwriting `caller` pointers, so defensive code needs to avoid this threat by never calling non-defensive functions. This is not an additional restriction because, as discussed above, it is also needed to avoid dependence on non-defensive state.

4.2 DJS: formal definition

We now formally define DJS, our defensive subset of JavaScript, by a combination of syntactic restrictions and type checking. The "strict mode" of ECMAScript 5 (ES5S) guarantees some properties useful for defensiveness, such as lexical scoping or safe binding of `this`, but for compatibility with a large number of deployed web browsers which only support ECMAScript 3, we do not rely on ES5S features. [1] DJS code that does not use the `with` statement is valid ES5S code.

Syntax. The syntax of DJS is reported in Figure 2. In this section we present only the core of DJS, omitting derivable constructs such as for example for loops, switch, and the increment operator. Tokens prefixed by @ refer to the lexical categories of the standard ECMAScript syntax.

A DJS program is essentially an arbitrary DJS function on strings wrapped in a stub that hides the function inside the variable "_" of its closure, and dynamically typechecks the input parameter x. For simplicity, functions have all variables declarations at the beginning and a single return statement at the end.

Statements and expressions are standard. As often required to allow static analyses, there is no general computed property access `e[e]` in the syntax. However, we include *dynamic accessors* that allow numeric, within-bound property access for arrays and strings using built-in dynamic checks, such as `x[(e>>>0)%x.length]`. If `x` is a non-empty array and `e` is known to evaluate to a number, then `e>>>0` is a positive integer, and the expression `(e>>>0)%x.length` evaluates to a value in $[0..\texttt{x.length} - 1]$. Furthermore, if `x.length` is known to be a power of 2, `e&2^n-1` also ensures that the access is not out of bounds; because of its conciseness and small overhead this check is favoured in practice.

Types. DJS types and their subtyping relation are defined in Figure 3. Function types are standard, and are distinct from method types that single out a variable ρ for the type of the `this` parameter. In order to avoid accessing ar-

[1] On the other hand, our type system gives stronger guarantees than ES5S even on lexical scoping and the safe binding of `this`. For example, it prevents exceptions and ensures that methods are not called outside objects, whereas in ES5S if a method is called outside an object, `this` is bound to `undefined` and access to its properties will cause an exception. Similarly, write access to global variables from a strict function will cause an exception only if the variable is not defined, whereas our type system prevents access to all non-local variables.

rays and strings beyond limits, array and string types are indexed by a lower bound on their length.

The type system of DJS is mostly monomorphic, 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. However, a standard width-subtyping relation $<:$ captures polymorphism in the length of arrays and strings, as well as the set of properties of objects.

Typing environments Γ reflect the nesting of the lexical scoping up to the expression that is being typed. Each scope typing frame Φ contains bindings of identifiers to types, and is annotated with `s` or `o` depending on whether the scope object corresponding to Φ 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. If the current frame is $[\Phi]_{\mathtt{s}}$, an unresolved identifier can be looked up in the rest of the typing environment Γ. If the current frame is $[\Phi]_{\mathtt{o}}$ then an unresolved identifier should be resolved by following a prototype chain, which we consider a type error.

Typing rules. The typing rules are detailed in figure 4. The rules for literals are standard. In particular, the rule for typing object literals 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 accessor 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. This is to avoid following a prototype chain, potentially leading to non-defensive objects, in the case when the top-level scope is an object frame and an identifier is not found locally. Rule With illustrates the case when an object frame is added to the typing environment. The rules for statements just propagate typing to sub-terms. The rules for binary and unary operators are standard. The -Cast rules type safe type-casts that do not involve executing user-defined type conversion code. 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 typecheks 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 a valid type of the `this` parameter ρ. In particular, ρ must be such that method l has a function type compatible with the potentially more general type given to l in the original object. The current type system does not allow recursive method calls. In practice, we can always infer the ρ type for non-recursive method invocations.

4.3 Conformance checker tool

A JavaScript program is considered valid DJS if it is accepted by our conformance checker. The tool automatically type checks an input script or provides informative error messages if the script fails to typecheck. Figure 5 shows a screenshot with a type error and then the correct inferred type.

Type inference. 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, the tool implements a restricted variant of Hindley–Milner inference where objects, arrays and strings are polymorphic in the number of fields. For example, the generalized type for the function `function f(x){return x[2]}` is $\exists\tau, \forall n \geq 3, [\tau]_n \rightarrow \tau$. Notice that the use of an existential instead of a universal quantifier in front

of τ shows that in our system there is no generalization for function types. This is due to the well-known problem of mutable references breaking functional polymorphism. Thus, if the type inference processes the term `f([1,2,3])`, unification will force $\tau = \mathsf{number}$, and any later attempt to use `f(["a","b","c"])` will cause a unification error.

The unification of polymorphic object types 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, non polymorphic type $\{x_i : T_i\}_{i\in[1..n]}$. Unification of a polymorphic object type $\{X\}$ with the non polymorphic type $\{x_i : T_i\}_{i\in[1..n]}$ checks that $X \subseteq \{x_i : T_i\}_{i\in[1..n]}$.

Our tool uses type inference as a heuristic, and relies on the soundness of the type checking rules of Section 4.2 for its correctness. Here we do not give further details of our inference and unification algorithms, which are standard. We invite interested readers to refer to our implementation of type inference, which we claim is polynomial in the size of the input expression to typecheck.

4.4 Limitations and extensions of DJS

Because of its very specific goals, DJS doesn't offer many features JavaScript programmers normally rely on, most notably:

- prototype functions of base objects such as strings (charCodeAt, indexOf, substr...), arrays (map, forEach, slice...) or regular expressions (exec, test...);
- access to library and DOM functions (eval, encodeURIComponent, XMLHttpRequest, createElement...);
- property enumeration `for(i in o)`;
- constructors and prototype inheritance (new, instanceof...);
- extensible arrays and objects and arbitrary property access (new objects can only be created with a fixed shape);
- user exceptions.

While most of the above features break defensiveness (sometimes subtlety: `function(){return "a" in {}}`), some are in fact compatible with defensiveness but were not included in core DJS, either because they would significantly complicate the type system, or because their safety depends on more restrictive assumptions about the behavior of non-defensive programs.

Constructors. Since any literal function `f` has a prototype property, it is possible to assign a literal object to `f.prototype` and instantiate the constructor using `new f()`. While this pattern is widespread in JavaScript, it does not offer any additional expressiveness, since prototype methods can be replaced with functions that accept the target object as an argument. For simplicity we do not include constructors in DJS although they can be used safely in this manner.

User exceptions. Our type inference tool partially supports user exceptions using a special binding `@exception` in the typing environment that gets introduced when entering a `try` block or function. The `throw` statement is equivalent to an assignment to this binding, and the `catch(e)` block exposes this internal binding through the identifier `e`. The types of functions and methods are extended to support an optional exception type along with their return types. In a try block, any function or method that is called must have an optional exception type that can unify with `@exception`. Calling a function with an exception type outside a try block will propagate this type to its parent function, or cause inference to fail at the top level.

By treating `return` statement like exceptions, type inference also allows functions to return prematurely rather than enforcing DJS's strict function syntax.

Extensible objects. In core DJS arrays and objects cannot be extended by adding new properties. In general, this problem is not straightforward. For example, we found that the mechanism to extend JavaScript arrays by setting their `length` property is broken: while

it correctly removes properties when decreasing the length, it does not create new ones when the length is increased, so accesses beyond the previous length value are resolved non-locally through the prototype chain (as a result, we allow only read access to the length property of arrays in DJS).

Still, extensible arrays are very useful for example when parsing a JSON string containing arrays of unknown length. For this reason, we implemented the recursive list type `'a list = null + {head:'a, tail:'a list}` in the type checker, and provided a defensive list library LArray.js for DJS. This solution is inspired by the encoding of recursive algebraic data types as nested objects with name and parameters properties proposed in [16]. We do not use this encoding in full generality as it conflicts with our goals about automatic inference (the programmer would have to declare algebraic types in annotations) and performance (ML-like lists introduce a linear access time plus a method call overhead, compared to a direct property access). Moreover, since our syntax is a subset of JavaScript, it is not natural to define new type constructors since they share the syntax of literal objects.

A more general solution to safely extend objects is based on the observation that watches, getters and setters can only be defined by giving an explicit property name to monitor. Thus, if every property name within a defensive script is prefixed by a long, one-time random prefix, the attacker cannot poison the base prototypes to intercept access to the properties. The property accessor `e[PREFIX+p]` can be safely used for any object `e` and property `p`, existing or new. It is also possible to use `for(x in o)` to check which properties belong to the script (those that start with the prefix) and which were defined in the underlying prototype. In the DJS library DObject.js we build on this intuition to provide extensible objects, arrays and strings along with their default methods (concat, split, push etc.). Despite the obvious benefits of this solution, its safety depends on a strong assumption about the capability of arbitrary JavaScript code. While we believe this assumption to hold for current browsers, it is well possible that an extension of JavaScript may break this property. For example, mechanisms such as `__noSuchMethod__` suggest that an object-wide property access method may be introduced in ECMA6, where it appears as *catch-all proxies* in current drafts.

Dynamic accessors. The core DJS syntax includes two direct dynamic accessors for arrays and strings. In many cases though, strings and arrays are scanned linearly, so we include in the full language also the pattern `for(i=0;i<x.length;x++){s}` where `x` is a variable of type string or array and `i` does not appear on the left hand side of any assignment within `s`. We rely heavily on this construction in our defensive code for its ubiquity and complete lack of overhead.

5 Defensiveness

We now present the formal argument of defensiveness for DJS. The main notations, mostly borrowed from [17], are reported in Figure 6. The execution of a JavaSCript program P in the initial heap H (the memory), with current scope chain L (the stack) is denoted by

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

where H' is the final memory, r is the returned value and Σ is a trace of function calls. A heap cell $(l, x) \mapsto v$ denotes that the object in memory at location l has a field x storing value v. Heaps are obtained by composing cells with the commutative and associative juxtaposition operator $*$. A scope chain is a list of addresses of objects in the heap that reflects the nesting of lexical scopes at a given execution point. Traces record the address, actual parameters and return values of every function call executed during a computation, and will be useful to discuss defensiveness formally.

5.1 Attacker model

The informal goal of defensiveness is to protect the behavior of sensitive JavaScript functions that will be invoked by arbitrary adversarial

code, when loaded in a JavaScript engine where adversarial code has already run.

Even if we had *the* formal semantics of JavaScript, using the execution of arbitrary programs in that semantics as a way to capture attackers would be a fragile solution due to implementation differences between the browsers, undocumented features, and foreseeable changes in future versions of the language.

Instead, we propose a general attacker model inspired by the successful Dolev-Yao attacker [14] used in the analysis of cryptographic protocols, where *the network is the attacker*. In JavaScript, we claim that *the memory is the attacker*. We model known and unknown semantics of arbitrary malicious code by allowing the attacker to arbitrarily change one (well-formed) JavaScript memory into another. In this spirit, our use of the notation $H, L, P \longrightarrow H', r, \Sigma$ should be taken to signify that H', r, Σ are respectively the final memory, the result and the execution trace of an arbitrary program P that may use non-standard or undocumented features of JavaScript, and that is executed according to any of the main existing JavaScript implementations.

Without making any further assumption, the attacker is just too powerful and nothing certain can be said about the properties of a system. Therefore, in the Dolev-Yao case, the attacker is restricted by the reasonable assumption that it is not able to guess secret cryptographic keys. Similarly, in the JavaScript case, we need to make the reasonable assumptions (satisfied by all known JavaScript implementations) that the attacker cannot manufacture pointers to existing memory locations to which it does not have access, and that it cannot peek inside function scopes.

Let an *attacker memory* be any JavaScript heap that includes at least all the native objects and the constant memory locations on which the semantics relies. In other words, an attacker memory is just an arbitrary memory in which JavaScript programs can be run.

Let a *user memory* be a heap whose domain only includes dynamically allocated (fresh) locations. A user memory denotes local memory allocated during the execution of some user code. A JavaScript program cannot in general be executed just in user memory, because for example it would contain the definition of native objects, prototype and scope objects that are necessary for safe execution.

In order to limit the power of the memory attacker, assume that in any reasonable JavaScript semantics, the execution of an arbitrary program in an attacker memory extended with a fresh user memory leads to a final memory that can be partitioned in two disjoint components, where one of the two is the unchanged fresh user memory. This assumption, formalized below, captures the fact that the attacker cannot forge pointers.

Assumption 1 (Memory safety)**.** Assume P is an arbitrary JavaScript program, H is an attacker memory and L an scope chain for H. For any user memory H_1,

$\forall H', r, \Sigma.\ \Sigma_{|call(H_1)} = \emptyset \Rightarrow$
$(H \circledast H_1, L, P \longrightarrow H', r, \Sigma \quad \Rightarrow \quad \exists H_2.H' = H_2 \circledast H_1)$,
where $call(H_1)$ denotes the calls to functions in H that contain a pointer to H_1 in their internal lexical scope. □

$H \circledast H'$ means that H and H' have disjoint domains and that H does not contain pointers to objects in the domain of H'. Note that a function in H can still contain in its internal scope chain a pointer to an activation record defined in H', so Assumption 1 holds only under the hypothesis that P does not trigger a call to a function that has in its scope an object from H_1.

5.2 Formal defensiveness

We now formally define the two properties that capture our intuitive notion of defensiveness from Section 5.

Definition 1 (Behavioral secrecy)**.** The function wrapper $E[-]$ maintains the *behavioral secrecy* of a function expression F if an arbitrary script Q cannot tell the difference between the script $x = E[F]$ and the script $x = E[F_{\mathsf{ID}}]$, where F_{ID} = `function(y){return y}`, without calling F or F_{ID}.

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

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

Definition 2 (Independence)**.** The function wrapper $E[-]$ preserves the *independence* of a function expression F if, whenever it is called with the same parameters, it returns the same results. $\forall H, L, P_1, P_2, H_1, H_2, r_1, r_2, \Sigma_1, \Sigma_2.$
$\big(H, L, x = E[F]^l; P_1 \longrightarrow H_1, r_1, \Sigma_1$
$\wedge\ H, L, x = E[F]^l; P_2 \longrightarrow H_2, r_2, \Sigma_2\big)$
$\Rightarrow \big(\Sigma_1 \sim_l^c \Sigma_2 \Rightarrow \Sigma_1 \sim_l \Sigma_2\big)$

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

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

5.3 Type safety

Before arguing that DJS can be used to define functions that enjoy Behavioral secrecy and Independence, we establish a stronger *type safety* property for the whole subset. This requires a formal semantics of a JavaScript fragment that covers at least DJS; for our proof, we adapt the operational semantics described in [17], which we denote by $H, L, P \xrightarrow{\mathsf{DJS}} H', r$. [3] However, since DJS uses only few and basic language features, we claim that our formal results do not depend on the specific choice of the semantics, and are robust to reasonable revisions of JavaScript. We formalize this intuition as an explicit assumption.

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

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

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

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

The proof of this Theorem is reported in Appendix A. Besides the soundness of our type

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

[3] While we made an effort to keep this section self-contained, a detailed exposition of formal JavaScript semantics goes beyond the scope of this paper, and we address the reader to [17] for further details.

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

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

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

The proof of this Theorem is reported in Appendix A.

6 Defensive Libraries

In the previous sections we detailed various restrictions we imposed on DJS to enforce defensiveness. Next, we show that DJS is still expressive enough to program security-critical components. In particular, we describe two defensive libraries for cryptography and data encoding.

6.1 DJCL: the Defensive JavaScript Crypto Library

We took as our starting point two reputable and widely used JavaScript libraries for cryptography: SJCL [31] (covering hashing, block ciphers, encoding and number generation) and JSBN (covering big integers, RSA, ECC, key generation). We converted a part of these libraries to DJS and verified it with our conformance checking tool. Our implementation covers the following primitives: AES on 256 bit keys, SHA-256, HMAC, RSA on moduli up to 2048 bits using OAEP for encryption and PSS for signature.

In general, cryptographic algorithms are well suited to the defensive style, because they use standard arithmetic operators on bitstrings and are usually self-contained. Still, when manipulating arrays of bitstrings, we needed to add dynamic checks to many array accesses. These checks introduce a performance overhead compared to non-defensive implementations. Interestingly, however, we found that the lack of dynamic object extension in defensive code lets JIT compilers better optimize DJS programs, yielding a speedup that more than offsets any overhead from dynamic checks.

We evaluated the performance of DJCL using the `jsperf` benchmark engine on Chrome 23, Firefox 16, Safari 5.1 and IE 9. We found our implementation of AES to be about as fast as SJCL's on all browsers, while our version of the SHA-256 block function performed slightly better on some (from similar performance in IE to up to 50% faster in Chrome). For RSA, we found our version of modular exponentiation to perform the same as `jsbn`. In practice, DJS performance depends also on the significant overhead for encoding and converting strings to input words of block functions. Depending on the encoding of the input (UTF-8, base64, hex string), high-level operations like CBC encryption or SHA-256 hashing can become significantly (up to 80%) slower than non-defensive code.

6.2 DJSON: Type-Safe JSON Serialization and Parsing

Implementing encodings in DJS is far more challenging than cryptography. The problem is that defensive code cannot rely on String.charCodeAt, encodeUriComponent, and String.fromCharCode, since they may have been manipulated by an attacker, but these functions are needed to encode and decode Unicode characters. For ASCII characters, we store the entire encoding table in a literal object and use a special dynamic accessor (x.length==1 && x<="c"? t[x] : d) that forces the object t to contain all characters from the null byte to c; still, this method is up to a hundred times slower than charCodeAt, which is heavily optimized. For Unicode characters, storing a literal table is infeasible due to its size. Instead, we rely on the built-in lexicographic order on strings to store the complete UTF-8 alphabet in a string and use binary search to find the code of a given character. We offer special optimizations for the ASCII subset.

In addition to code point lookup, the decoding of unicode characters into the corresponding byte sequences must be performed manually. Because the usual method of using `unescape` and `encodeURIComponent` is actually quite slow, this change can increase performance.

In most of our applications, the input string of a DJS program represents a JSON object. While serializing well-typed JSON objects poses no problems, parsing serialized JSON strings generally requires the ability to create extensible objects. Instead, we rewrite JSON.parse defensively by requiring two additional parameters to parse a string: the first is a description of the fields and types that are expected from the input string, the second is a preallocated object of the right shape that will be filled by DJSON.parse. In principle, our type inference tool should be able to automatically generate the appropriate type schema and object literal, but for now we require this information from the programmer. Any type mismatch or missing field during will result in a parsing error.

As for serialization, it suffers from the inability to differentiate properties of an object and properties from its prototype without `Object.hasOwnProperty`: both `in` and `for(i in o)` will accept properties set in the object's prototype. Like before, we must known at runtime the type of the object being serialized. Once again, this type is known by the type system, but to avoid any compilation or preprocessing step of DJS code we again require this type to be provided by the programmer.

6.3 JOSE: libraries for JSON Cryptography

Combining DJCL and DJSON, we implemented a family of emerging JSON cryptography standards, including JSON Web Tokens (JWT), JSON Web Signatures (JWS), and JSON Web Encryption (JWE). Our defensive JOSE library is fully interoperable with other implementations in PHP. It is able to exchange encrypted and signed tokens with them, while keeping its keys secret from its hosting page. The performance of our library is comparable to that of PHP (and it is noticeably faster in Chrome.) We rely on this library to build secure communications for our defensive applications.

7 Applications

Using DJS and the defensive libraries, we program several defensive patterns and use them to build applications: i) We show how to distribute scripts with embedded secrets so that the pages into which they are loaded cannot learn these secrets. In particular, we show how to distribute cryptographic keys using this mechanism. ii) We show how to use JWE to program secure channels between defensive scripts and other entities such as websites (via AJAX), frames (via postMessage), or other defensive scripts (via function calls). iii) We show how to implement fine-grained access control for local storage using encryption.

7.1 Secure Script Server and Encrypted AJAX

Since defensive scripts can be loaded from any origin, we rely on the concept of a *secure script server* to deliver the defensive scripts of a given component provider. Concretely, this is a web service running in a trusted, isolated origin (a subdomain like `defscripts.facebook.com`) that accepts as parameters a script name and a target origin.

It first checks that the target origin is registered with a publicKey, and that the requested script exists in the defensive script store. It authenticates that the target origin is the same as the hosting origin by verifying the Origin header on the request. When this header is not available, we can fall back to a client-side origin verification method based on encrypted AJAX, which is described below.

The next step is to generate a fresh hostKey. This will be used to encrypt communication with the target origin's server and be wrapped using publicKey.

The last step is to retrieve the sessionKey, which should be the same in all subsequent requests during the same browsing session. We rely on a cookie-based session to keep track of this key; if no valid session cookie is provided in the request, a new session is created with a fresh sessionKey.

Finally, the requested script is returned with the authenticated origin, publicKey, hostKey and sessionKey embedded defensively into the script.

A simple use case for the embedded keys is to perform an encrypted AJAX query to the host page's server. The following example demonstrates how to use this pattern to get the current time:

```
(function(){
 var origin = 'x.com';
 var RPC = (function() {
   var publicKey = "...", hostKey = "...";
   var DJCL = {...}, state = 0;
   var _ = function(x) {
    state = 1-state;
    if(!state) // Request
     return DJCL.JWE.create(x, hostKey,
          publicKey);
    else // Response
     return DJCL.JWE.parse(x, hostKey, publicKey
          );
   }
   return function(x) {
    if(typeof x=='string') return _(x)
   }})();
 return function() {
  var xhr = new XMLHttpRequest();
  var d = RPC("getTime()");
  xhr.open("GET", "http://"+origin+"/RPC?d="+d)
       ;
  xhr.onload = function() {
   alert('The time is: '+RPC(xhr.response));
   };
  xhr.send();
 }
})()()
```

If the requested origin could not be authenticated by the script server, a similar AJAX request containing the encrypted hostKey under publicKey and returning the origin encrypted under hostKey and signed with the host private key can be used to recover the genuine origin.

7.2 Securing Login with Facebook

Our survey of OAuth 2.0 implementations in Section 3 revealed many different ways that access tokens could be leaked to malicious websites. While some of our attacks exploited bugs that should legitimately be fixed, we believe that a secure login component cannot rely on the host website not having any vulnerabilities on any of its webpages. Instead we propose a new defensive architecture for Facebook login that makes it robust even in the face of such vulnerabilities.

Restricting Access Tokens. As a first step, we propose that the Facebook SDK should never expose the access token directly to the hosting webpage where it could then be stolen by an untrusted script. As a proof-of-concept, we show how to achieve this using iframes and cryptography.

Suppose the website W registers the origin https://fb.W.com at Facebook. It then loads the entire Facebook SDK within a frame sourced from https://fb.W.com. This frame can obtain the access token by calling FB.login and then call FB.api, however the parent window has no direct access to the token or these functions. Instead, the iframe at https://fb.W.com offers a

limited API to the parent window. The parent may send a message to request FB.login() or FB.api(), but these messages must be authenticated using a host key that is only made available to defensive scripts loaded from a script server at `https://secure.W.com`. If the parent asks for the token by sending a message requesting FB.getAccessToken(), the frame returns the token encrypted under the host key.

As a consequence of these modifications, we are able to wrap the full Facebook JavaScript SDK within a defensive component that never reveals access tokens to untrusted scripts and only allows trusted scripts to call the Facebook API. As a further enhancement, the wrapper could also filter requests to FB.api and only enable a subset of the capabilities granted by the access token, a feature that cannot be offered by that current Facebook SDK.

Two Factor Authentication. As a second step, we show how to use a bookmarklet to obtain second factor of authentication for the Facebook Login component. One weakness of Facebook's implementation of the OAuth protocol is that once a user has authorized a website, the website is able to request a token for the user on all subsequent visits, even if the user did not want to identify herself on every visit. Her only option to regain her privacy is to explicitly de-authorize the website on Facebook. Instead, inspired by bookmarklets used by password managers, we propose a new mechanism that Facebook could use to verify that the user indeed meant to login to a website.

To use Facebook login, users download and install a JavaScript bookmarklet on their browser that contains its own identifier and secret key. When the bookmarklet is clicked on a webpage, it installs a defensive script on the web page that is willing to countersign, with its secret key, any FB.login request generated by the hosting webpage. The JavaScript SDK (hosted within the iframe) now requires that the postMessage for FB.login be signed both using the host key for the current hosting page and the bookmarklet key.

Hence, the current webpage will be unable to log the user into Facebook, unless the user clicked on the bookmarklet. To ensure that the website does not cache the bookmarklet's response and replay it later, we implement a challenge response mechanism, where the bookmarklet signature must include a nonce provided by the Facebook SDK.

This bookmarklet-based solution significantly increases the security of Facebook login but would not have been possible without our ability to program the bookmarklet defensively and embed our defensive HMAC-SHA256 library in it, since bookmarklets execute in the context of the current active page and are thus vulnerable to rootkitting. Our bookmarklet is able to fit the relevant parts of DJCL and DJSON into a self-contained defensive script under 3KB.

7.3 Securing Client-side Encryption for ConfiChair

We propose two defensive mechanisms to improve the security of web applications that use client-side encryption APIs for cloud storage.

For applications like Helios, Wuala, and ConfiChair, that use a Java applet for cryptography, we propose to replace the applet with DJCL. This has the dual advantage of significantly increasing the performance of the application (up to 100 times faster on large inputs) and of reducing the attack surface by removing the Java runtime from the trusted computing base.

For applications that need to store session keys or long term keys in the browser, we propose a new encrypted local storage mechanism leveraging defensive cryptography. We showcase these mechanisms by making ConfiChair more resilient to XSS attacks.

We set up a secure script server at `https://secure.confichair.org` that serves a script that first verifies that it has been loaded on ConfiChair (using one of the two origin authentication methods we described), and then exposes a cryptographic API with the following functions:

```
derive_secret_key:
  (input:string, salt:string) -> key:string
encryptData, decryptData:
  (data:string, key:string, IV:string) -> string
```

Defensive Program	Lines of Code	Typechecking Time
DJCL	865	211ms
DJSON	489	156ms
JOSE	162	53ms
Encrypted AJAX	14	7ms
Facebook Login Wrapper	135	42ms
ConfiChair Crypto API	80	31ms

Table 2: Evaluation of our DJS codebase

```
base64_encode, base64_decode:
  (input:string) -> string
encryptKeypurse:
  (keypurse:json string) -> json string
```

Compared to the original design, when the keypurse is decrypted, all of its keys are wrapped with the sessionKey set by the script server. When a decryption operation is requested, the defensive API will transparently unwrap the key, but never directly expose it to the page. In addition, when the keypurse is reencrypted using the master key derived from the passphrase at login, this master key is also wrapped under the sessionKey before being written to localStorage. We use authenticated encryption, combining AES-256-CBC with HMAC-SHA256, to prevent tampering of the keypurse data.

The only way for an attacker to get access to the sessionKey (besides breaking defensiveness) is to recover the secure HTTP-only cookie set by the secure script server to track the sessionKey. Additional components loaded from different script servers may also store data in localStorage wrapped under their own sessionKey without exposing the stored content to other scripts running on the page.

Our modifications to the ConfiChair website amount to replacing its Java applet with our defensive API and rewriting two lines of code on the Login page. The rest of the website works without any modification while enjoying a significanly improved level of security. An XSS attack on the conference page may be able to read articles by calling the defensive API, but can no longer steal keys or passwords.

8 Security Evaluation

We have presented several applications written in DJS that use our defensive cryptographic library to program security functionalities within untrusted origins. How do we know that these applications achieve their security goals?

Defensiveness only guarantees that the functionality of defensive scripts cannot be tampered with, it does not say that the script will not leak its secrets to the hosting webpage, by exposing them in its exported API, for example. Similarly, defensiveness does not guarantee that our cryptographic library is implemented correctly or that our applications use cryptographic functions correctly.

For our libraries, we rely on careful inspection and extensive interoperability testing to convince ourselves of their correctness. For our applications, we try to use standard cryptographic constructions such as authenticated encryption for local storage, authenticated and encrypted RPCs for secure AJAX, and HMAC for bookmarklet authentication. Our only secrets are encryption keys injected into our scripts by the secure script server, and by inspection we can show that these keys are only used for cryptography.

Still, for more complex applications and cryptographic protocols, we advocate that the design of the security component be verified against a realistic model of browser security mechanisms and web-based attackers. We used the WebSpi model of [6] to model and verify the original designs and our modifications to Facebook login and the ConfiChair website. By analyzing our models using the protocol verifier ProVerif, we were able to find many of the attacks of Section 3. Conversely, we analyzed our modified designs and did not find any attacks among the set of attacks modelled in WebSpi. This positive result should not be taken as a formal theorem, since the browser model of WebSpi is incomplete and may miss many attacks. Instead, we treat this verification as providing additional confidence in our application design.

9 Conclusions

Client-side security components have come into popular use because in many multi-party web interactions, such as single sign-on, there is no

single server that can be trusted to enforce all the necessary security checks. Instead, we have come to rely on the browser to tie these interactions together using cookies, HTTP redirections, frames, and JavaScript.

Several emerging web security standards aim to give the browser more fine-grained control on what web compositions it should allow. The Web Cryptography API (WebCrypto) provides a standard interface to browser-based cryptography and key storage. Content Security Policy (CSP), the Origin header, and Cross-Origin Request Sharing (CORS) tell the browser what external content is allowed to be loaded onto a webpage. ECMAScript Version 5 Strict Mode defines a safer subset of JavaScript meant to be enforced by the browser.

Our approach is complementary to these new standards, since their guarantees only extend to trusted websites and not to tampered environments, which will still need to be defended against. When their implementations are stable and widespread, we may be able to allow more programming constructs in DJS while retaining its strong security guarantees. Meanwhile, DJS can already be used with current web applications and can significantly improve their security.

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

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

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

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

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

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

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

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

⟨*unop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`!`’ | ‘`{~}`’

Figure 2: DJS Syntax.

Types.

$\langle\tau\rangle ::=$ number | boolean | string | undefined — Base types
| $\tilde{\tau} \to \tau$ — Function
| $\tilde{\tau}[\rho] \to \tau$ — Method operating on properties ρ
| σ — Indexed type
| σ^* — Fixed index type

$\langle\sigma\rangle ::= \rho \mid [\tau]_n, n \in \mathbb{N}$ — Array of length n

$\langle\rho\rangle ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ — Object with fields $x_1 \cdots x_n$

Subtyping.

$$\tau <: \tau \quad \forall\tau$$
$$\sigma^* <: \tau \quad \text{when} \quad \sigma <: \tau$$
$$[\tau]_n <: [\tau']_m \quad \text{when} \quad m \leq n \wedge \tau <: \tau'$$
$$\{x_i : \tau_i\}_{i\in I} <: \{x_j : \tau'_j\}_{j\in J} \quad \text{when} \quad J \subseteq I \wedge \forall i \in J, \tau_i <: \tau'_i$$
$$\tilde{\mu}_1 \to \nu_1 <: \tilde{\mu}_2 \to \nu_2 \quad \text{when} \quad \nu_1 <: \nu_2 \wedge \tilde{\mu}_2 <: \tilde{\mu}_1$$
$$\tilde{\mu}_1[\rho_1] \to \nu_1 <: \tilde{\mu}_2[\rho_2] \to \nu_2 \quad \text{when} \quad \rho_2 <: \rho_1 \wedge \tilde{\mu}_1 \to \nu_1 <: \tilde{\mu}_2 \to \nu_2$$

Environments.

$\langle\kappa\rangle ::=$ s | o — Scope kind

$\langle\Phi\rangle ::= \varepsilon \mid \Phi, x{:}\tau$ — Scope frame

$\langle\Gamma\rangle ::= \varepsilon \mid \Gamma, [\Phi]_\kappa$ — Typing environment

Figure 3: DJS types, subtyping and environments.

```
$ ./jsdef --check
greet = function(x){var i,res;
 for(i=0,res=[]; i<x.length; i++)
  res.push("Hi "+x[i]);
 return res}
Cannot type the following expression at file <stdio>,
 line 2:5 to 2:6:
i = 0
type <number> was expected but got <undefined>.
$ ./jsdef --check
greet = function(x){var i=0;
 for(i=0; i<x.length; i++)
  x[i]="Hi "+x[i];
 return x}
Typing was successful, program is defensive.
--- Free variables ---
greet: (string+ array[>=1]) -> string+ array[>=1]
$
```

Figure 5: Screenshot of the type checker output: first an informative error, then the inferred type.

Semantics.

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

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

Heaps, scope chains, programs and traces.

$\langle H\rangle ::=$ emp — Empty heap
| $H * (l, x) \mapsto v$ — Heap cell, $(l, x) \notin dom(H)$

$\langle L\rangle ::= []$ — Empty scope chain
| $l : L$ — Scope frame l on top of L

$\langle P\rangle ::= \ldots$ — Arbitrary JavaScript program

$\langle\Sigma\rangle ::= \varepsilon$ — Empty trace
| $l(\tilde{v}) \to v : \Sigma$ Call to l with parameters $\tilde{v}$ returned v

Heap domain and codomain, trace projection.

$$dom(\texttt{emp}) = \emptyset \quad cod(\texttt{emp}) = \emptyset$$
$$dom(H_1{*}H_2) = dom(H_1) \cup dom(H_2) \quad dom((l, x) \mapsto v) = \{(l, x)\}$$
$$cod(H_1{*}H_2) = cod(H_1) \cup cod(H_2) \quad cod((l, x) \mapsto v) = locations(v)$$
$$(l(\tilde{v}) \to v : \Sigma)_{|l} = l(\tilde{v}) \to v : \Sigma_{|l} \qquad \emptyset_{|l} = \emptyset$$
$$(l'(\tilde{v}) \to v : \Sigma)_{|l} = \Sigma_{|l} \text{ if } l \neq l'$$

Notation.

$$H_1 \boxplus H_2 \triangleq H_1 * H_2 \text{ where } cod(H_1) \cap dom(H_2) = \emptyset$$
$$l \mapsto \{\mathtt{x}_1{:}\, v_1, \ldots, \mathtt{x}_n{:}\, v_n\} \triangleq (l, x_1) \mapsto v_1 {*} \ldots {*} (l, x_n) \mapsto v_n$$

Expression contexts.

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

Well-formedness of user memory.

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

Figure 6: Semantics notation.

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

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

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

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

$$\text{Scope}\frac{\Phi(x) = \tau}{\Gamma, [\Phi]_\kappa \vdash x : \tau} \qquad \text{RecScope}\frac{x \notin dom(\Phi) \quad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_\mathsf{s} \vdash x : \tau} \qquad \text{With}\frac{\Gamma \vdash e : \{\tilde{x} : \tilde{\tau}\} \quad \Gamma, [\tilde{x} : \tilde{\tau}]_\mathsf{o} \vdash s : \mathsf{undefined}}{\Gamma \vdash \mathrm{with}(e)s : \mathsf{undefined}}$$

$$\text{If}\frac{\Gamma \vdash e : \mathsf{boolean} \quad \Gamma \vdash s, t : \mathsf{undefined}}{\Gamma \vdash \mathrm{if}(e)\ s\ \mathrm{else}\ t : \mathsf{undefined}} \qquad \text{While}\frac{\Gamma \vdash e : \mathsf{boolean} \quad \Gamma \vdash s : \mathsf{undefined}}{\Gamma \vdash \mathrm{while}(e)\ s : \mathsf{undefined}}$$

$$\text{Block}\frac{\Gamma \vdash s_i : \mathsf{undefined} \quad i \in [1..n]}{\Gamma \vdash [\{]s_1; \ldots; s_n[;][\}] : \mathsf{undefined}} \qquad \text{Arit}\frac{\Gamma \vdash e_1 : \mathsf{number} \quad \Gamma \vdash e_2 : \mathsf{number} \quad \circ \in \{+, -, *, /, \%, \&, |, \hat{}\}}{\Gamma \vdash e_1 \circ e_2 : \mathsf{number}}$$

$$\text{Assign}\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \tau} \qquad \text{Concat}\frac{\Gamma \vdash e_1 : \mathsf{string} \quad \Gamma \vdash e_2 : \mathsf{string}}{\Gamma \vdash e_1 + e_2 : \mathsf{string}}$$

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

$$\text{UnOp}\frac{\Gamma \vdash e : \mathsf{number} \quad \circ \in \{-, \sim\}}{\Gamma \vdash \circ e : \mathsf{number}} \qquad \text{BoolCast}\frac{\Gamma \vdash e : \tau}{\Gamma \vdash !e : \mathsf{boolean}}$$

$$\text{NumCast}\frac{\Gamma \vdash e : \mathsf{string}}{\Gamma \vdash +e : \mathsf{number}} \qquad \text{StrCast}\frac{\Gamma \vdash e : \mathsf{number}}{\Gamma \vdash e + \text{""} : \mathsf{string}}$$

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

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

Figure 4: Typing rules.

References

[1] B. Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348, 2008.

[2] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript environments. WOOT'09, 2009.

[3] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. *Proceedings of the USENIX Security Symposium*, 2012.

[4] M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *POST*, pages 89–108, 2012.

[5] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

[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'09*, 2009.

[8] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *Proc. of 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] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *Workshop on Offensive Technologies (WOOT)*, 2012.

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

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

[13] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 748–759, 2012.

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

[15] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *BDSS'10*, 2010.

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

[17] P. Gardner, S. Maffeis, and Gareth D. Smith. Towards a program logic for JavaScript. POPL '12, 2012.

[18] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Protocol. IETF Internet Draft, 2011.

[19] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *CSF*, pages 3–18, 2012.

[20] Zhengqin L. and T. Rezk. Mashic compiler: Mashup sandboxing based on interframe communication. *CSF'12*, 2012.

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

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

[23] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. IEEE SP '10, 2010.

[24] J. Mickens and M. Finifter. Jigsaw: Efficient, Low-effort Mashup Isolation. In *USENIX Web Application Development*, 2012.

[25] P. Phung, D. Sands, and D. Chudnov. Lightweight self-protecting JavaScript. ASIACCS '09, 2009.

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

[27] 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.

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

[29] Matasano Security. Javascript cryptography considered harmful, 2010. `http://www.matasano.com/articles/javascript-cryptography`.

[30] Juraj Somorovsky, Andreas Mayer, Adolf Wrth, Jrg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *Workshop on Offensive Technologies (WOOT)*, 2012.

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

[32] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE SP'11*, 2011.

[33] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy*, pages 365–379. IEEE Computer Society, 2012.

[34] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *IEEE Symposium on Security and Privacy*, pages 465–480. IEEE Computer Society, 2011.

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

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

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

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

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

Figure 7: Typing rules for memory values

A Proofs

Well-typed memory values. For conciseness, in the definition of a well-formed user memory (Figure 6), we use the notation $\emptyset \vdash H(v){:}T$ for a heap H typing a value v with type T. In Figure 7 we give the formal definition of this relation, and we use the more explicit notation $H \vdash v{:}T$. We omit arrays and methods, because they do not differ in memory from objects and functions.

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

When we evaluate DJS code, we start from well-typed syntactic code and translate it to memory operations, where the values of the computations remain well-typed in the heap. In most cases, we will use the same notation for typing evaluated (memory) and non-evaluated (syntactic) expressions.

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

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

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

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

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

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

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

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

$$\frac{(l, x) \in \mathsf{dom}(H)}{\pi(H, l, x) \triangleq l} \qquad \frac{\begin{array}{c}(l, x) \notin \mathsf{dom}(H) \\ H(l, @proto) = l'\end{array}}{\pi(H, l, x) \triangleq \pi(H, l', x)}$$

Operational rules

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

(Member Access)
$$\frac{\begin{array}{c}H, L, \texttt{e} \xrightarrow{\gamma} H', l' \\ l' \neq \texttt{null}\end{array}}{H, L, \texttt{e.x} \longrightarrow H', l'{\cdot}\texttt{x}}$$

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

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

Figure 8: Select semantics rules from [17].

- *Scope lookup*: $\Gamma \vdash x : \tau$ can follow either from the Scope or RecScope rule from Figure 8. In the first case, we can decompose $\Gamma = \Gamma_0, [\Phi]_\kappa$ with $\Phi(x) = \tau$. Following the well-formedness hypothesis of $(H_1, L) \models \Gamma$ defined in Figure 6, we can decompose the memory $H_1 = H' * l \mapsto \{\dots, x : r, \dots\}$ and scope chain $L = l : L'$, knowing that $\Gamma \vdash H_1(r) : \tau$. Because $(l, x) \in \mathsf{dom}(H_1)$, $\pi(H_A \circledast H_1, L, x) = l$ and $\sigma(H_A \circledast H_1, L, x) = l$. Applying the Variable rule yields $H_A \circledast H_1, L, x \to H_A \circledast H_1, l \cdot x$. This proves the induction goal with $H_2 = H_1$ and $r = l \cdot x$.

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

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

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

  $$\begin{array}{lll} H_1 = H' & * & l \mapsto \{\dots, x : r, \dots\} \\ & * & l_1 \mapsto \{X_1\} * \cdots * l_n \mapsto \{X_n\} \end{array}$$

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

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

  using the second scope lookup rule, it follows that

  $$\begin{array}{lll} \sigma(H_A \circledast H_1, L, x) & = & \sigma(H_A \circledast H_1, l : L', x) \\ & = & l \cdot x = r \end{array}$$

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

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

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

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

  We also use the induction hypothesis on $\Gamma \vdash \tilde{e} : \tilde{\alpha}$ to run $H_A \circledast H_2^i, L, e_i \to H_A \circledast H_2^{i+1}, v_i$ where $H_2^0 = H_2$ and H_3 is the final heap after evaluating all the arguments. We are now ready to apply the function call rule using $\text{This}(l \cdot x) = l$ and $H_4 = H_3 * \text{act}(l, \tilde{x}, \tilde{v}, b, l)$, for which we claim:

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

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

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

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

  $$H_A \circledast H_1, L, o.x(\tilde{e}) \to H_A \circledast H_5, v'$$

  with $\Gamma \vdash H_5(v') : \tau$.

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

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

  with $\Gamma \vdash H_2(r) : \{(x_i : \tau_i)_{i \in I}\}$. Let $\ell = H_2(r)$, since ℓ has an object type and

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

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

By Definition 2, we need to show that for arbitrary H, L, P_1, P_2 and for Σ_1, Σ_2 such that $\Sigma_1 \sim_l^c \Sigma_2$,
(1) $\big(H, L, x = E[F]^l; P_1 \longrightarrow H_1, r_1, \Sigma_1 \quad \wedge$
(2) $H, L, x = E[F]^l; P_2 \longrightarrow H_2, r_2, \Sigma_2\big) \quad \Rightarrow$
(3) $\Sigma_1 \sim_l \Sigma_2$.

Following the reasoning for Lemma 1, if we have (1) and (2) then we also have that

$$H, L, x = E[F]^l; \longrightarrow H_E \circledast H_F, l, \Sigma_0$$

H_E is the new attacker memory containing in the lexical scope of l a pointer to the activation record l_D (allocated in H_F) of the wrapper function, and Σ_0 did not contain any call to l.

H_F is a user memory containing in l_F the function object corresponding to F. By the hypothesis $\emptyset \vdash F\colon \mathsf{string} \to \mathsf{string}$ and by type safety, we have $H_F \vdash l_F\colon \mathsf{string} \to \mathsf{string}$.

Let us consider the rest of the reductions
(4) $H_E \circledast H_F, L, P_1 \longrightarrow H_1, r_1, \Sigma_3$
(5) $H_E \circledast H_F, L, P_2 \longrightarrow H_2, r_2, \Sigma_4$
By definition of sequential composition, it must be the case that
(6) $\Sigma_1 = \Sigma_0 :: \Sigma_3$ and
(7) $\Sigma_2 = \Sigma_0 :: \Sigma_4$.
Since we assumed initially that $\Sigma_1 \sim_l^c \Sigma_2$, we need to argue that $\Sigma_1 \sim_l \Sigma_2$. Without loss of generality we can assume that P_1 has the form $P_{1,1}; x(y)_1; ...; x(y)_i; P_{1,n+1}$ and Σ_3 has the form

$$\Sigma_{1,1} :: l(v_{1,1}) \to r_{1,1} : ... :: l(v_{1,n}) \to r_{1,n} : \Sigma_{1,n+1}$$

where for all i, $(\Sigma_{P_{1,i}})_{|l} = \emptyset$. Similarly, P_2 has the form $P_{2,1}; x(y)_1; ...; x(y)_i; P_{2,n+1}$ and Σ_4 has the form

$$\Sigma_{2,1} :: l(v_{2,1}) \to r_{2,1} : ... :: l(v_{2,n}) \to r_{2,n} : \Sigma_{1,n+1}$$

and for all i, $(\Sigma_{2,i})_{|l} = \emptyset$. Each $P_{j,i}$ performs arbitrary computations that do not call function l, and then loads in a variable y the parameter $v_{j,i}$ for the invocation. Each $x(y)_i$ is the invocation of function l with $v_{j,i}$ obtaining result $r_{j,i}$, recorded in the trace Σ_i as $l(v_{j,i}) \to r_{j,i}$. Because of the $\Sigma_1 \sim_l^c \Sigma_2$ hypothesis, we can assume that $v_{1,i} = v_{2,i}$ for all i, so from here on we drop the indices j from each $v_{j,i}$.

Let $H_E^{1,1} = H_E^{2,1} = H_E$ and $H_F^{1,1} = H_F^{2,1} = H_F$, and let $P^{j,i}$ be the suffix of P_j defined as $P^{j,i} = P_{j,i}; x(y)_i; ...; x(y)_n; P_{j,n+1}$ and similarly for $\Sigma^{j,i}$. By inductive hypothesis, assume
(8) $H_F^{1,i} = H_F^{2,i}$
(9) $H_F^{1,i} \vdash l_F\colon \mathsf{string} \to \mathsf{string}$
(10 $H_E^{j,i+1} \circledast H_F^{j,i+1}, L, P^{j,i+1} \longrightarrow H_E^{j,n+1} \circledast H_F^{j,n+1}, r, \Sigma^{j,i+1}$
(11) $\Sigma^{1,i+1} \sim_l \Sigma^{2,i+1}$
At step i, by Assumption 1, $P_{j,i}$ transforms $H_E^{j,i} \circledast H_F^{j,i}$ in $H_E^{j,i+1} \circledast H_F^{j,i}$, where the (defensive) user memory $H_F^{j,i}$ does not change.

By (9) and type safety, since x evaluates to l and y evaluates to v_i, we have both

$$H_E^{1,i+1} \circledast H_F^{1,i}, L, x(y)_i \longrightarrow H_E^{1,i+1} \circledast H_F^{1,i+1}, r_{1,i}, l(v_i) \to r_{1,i}$$

$$H_E^{2,i+1} \circledast H_F^{2,i}, L, x(y)_i \longrightarrow H_E^{2,i+1} \circledast H_F^{2,i+1}, r_{2,i}, l(v_i) \to r_{2,i}$$

where in particular $r_{1,i} = r_{2,i}$ because in the type safety statement the result r is determined before the attacker memory H_A (here $H_E^{2,i+1}$). Moreover, by (8) and type safety we also have $H_F^{1,i+1} = H_F^{1,i+1}$.

Composing with the inductive hypothesis, we have

$$H_E^{j,i} \circledast H_F^{j,i}, L, P^{j,i} \longrightarrow H_E^{j,n+1} \circledast H_F^{j,n+1}, r, \Sigma_{j,i}$$

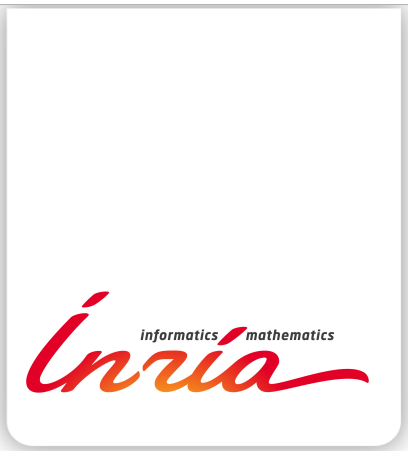
and combining with (11), we have $\Sigma^{1,i} \sim_l \Sigma^{2,i}$.

Hence,

$$H_E \circledast H_F, L, P_1 \longrightarrow H_E^{1,n+1} \circledast H_F^{1,n+1}, r, \Sigma_3$$

$$H_E \circledast H_F, L, P_2 \longrightarrow H_E^{2,n+1} \circledast H_F^{2,n+1}, r, \Sigma_4$$

and $\Sigma_3 \sim_l \Sigma_4$. This gives us (3) and (4), where $H_j = H_E^{j,n+1} \circledast H_F^{j,n+1}$. By composing with the wrapper execution and by (6) and (7), we obtain both (1),(2) and (3), concluding the proof. □



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399