Defensive JavaScript
Building and Verifying Secure Web Components

Karthikeyan Bhargavan[1], Antoine Delignat-Lavaud[1], and Sergio Maffeis[2]

[1] INRIA Paris-Rocquencourt, France
[2] Imperial College London, UK

Abstract. Defensive JavaScript (DJS) is a typed subset of JavaScript that guarantees that the functional behavior of a program cannot be tampered with even if it is loaded by and executed within a malicious environment under the control of the attacker. As such, DJS is ideal for writing JavaScript security components, such as bookmarklets, single sign-on widgets, and cryptographic libraries, that may be loaded within untrusted web pages alongside unknown scripts from arbitrary third parties. We present a tutorial of the DJS language along with motivations for its design. We show how to program security components in DJS, how to verify their defensiveness using the DJS typechecker, and how to analyze their security properties automatically using ProVerif.

1 Introduction

Since the advent of asynchronous web applications, popularly called AJAX or Web 2.0, JavaScript has become the predominant programming language for client-side web applications. JavaScript programs are widely deployed as scripts in web pages, but also as small storable snippets called bookmarklets, as downloadable web apps,[1] and as plugins or extensions to popular browsers.[2] Mainstream browsers compete with each other in providing convenient APIs and fast JavaScript execution engines. More recently, Javascript is being used to program smartphone and desktop applications[3], and also cloud-based server applications,[4] so that now programmers can use the same idioms and libraries to write and deploy a variety of client- and server-side programs.

As more and more sensitive user data passes though JavaScript applications, its confidentiality and integrity becomes an important security goal. Consequently, JavaScript applications rely on a number of security libraries for cryptography and access control. However, neither the JavaScript language nor its execution environment (e.g. the web browser) are particularly well suited for security programming. For example, to aid uniform deployment across different

[1] `https://chrome.google.com/webstore/category/apps`
[2] `https://addons.mozilla.org/`
[3] `http://dev.windowsphone.com/develop`
[4] `http://node.js`

A. Aldini et al. (Eds.): FOSAD VII, LNCS 8604, pp. 88–123, 2014.

browsers, JavaScript allows a number of core language primitives to be redefined and customized. This means that a JavaScript security library that may run alongside other partially-trusted libraries must take extra care so that its functionality is not subverted and its secrets are not leaked.

In this tutorial, we investigate approaches to build and verify JavaScript programs that implement security-criticial tasks, such as cryptographic protocols. Our programs must contend not just with the traditional network attacker, but also with a variety of web-specific attacks, such as malicious hosting websites and Cross-Site Scripting (XSS). In other words, not just the communication channel but even parts of the execution environment may be under the control of the adversary. We propose a typed subset of JavaScript, called Defensive JavaScript, that enables formal security guarantees for programs even in this threat model. Our language and verification results previously appeared in [12].

Many existing works investigate the security of formal models of web application protocols [3,17,8], but none of them can provide concrete security guarantees for JavaScript code. Still, we build upon these prior results (especially [8]) to develop our threat model and verification techniques. Another closely related line of work investigates the use of type-preserving compilers to generate JavaScript programs that are secure-by-construction [18,25]. We will focus only on language-based protections in JavaScript, but note that HTML-level isolation techniques may also be effectively used to separate trusted web security components from untrusted JavaScript [4].

In the rest of this section, we will seek to better understand the threat model and security goals of JavaScript security components through three examples.

1.1 Encrypted Cloud Storage Websites

Storage services (e.g. Dropbox) allow users to store their personal files on servers hosted within some cloud infrastructure. Since users often rely on these services to back up important files and share them across devices, the integrity and confidentiality of this data is an important security requirement, especially since the cloud servers may be under the control of a third party. Consequently, mainstream storage services typically encrypt user files before storing them in the cloud. A hacker who breaks into the cloud server to obtain the encrypted files would also need to steal the file encryption key from the storage service.

Some services, such as SpiderOak and Mega, seek to provide a stronger privacy guarantee to their users, sometimes called *host-proof* hosting — even if the storage service and its cloud servers are both hacked, the user's files should remain confidential. The key mechanism to achieve this goal is that a user's file encryption keys are generated and stored on the client-side; even the storage service does not get to see it, and so cannot accidentally leak it.

For example, to access their files stored on Mega, users visit the Mega website, which downloads and runs a JavaScript program in the browser. The program asks the user for a master passphrase, derives an authentication token and an encryption key from the passphrase, and sends the username and authentication token to the website. If the token matches the username, the web page allows the

user to download or upload encrypted files from a cloud server. The JavaScript program encrypts and decrypts user files upon request, using the encryption key derived from the master passphrase, but the key and the passphrase never leave the browser.

Hence, the storage service implements an application-level cryptographic protocol in JavaScript. This programming pattern is also popular with other security web applications such as password managers (more below) and with privacy-preserving websites like ConfiChair [5] and Helios [1].

The main threat to this design is that if the attacker manages to inject a malicious script into the website, that script will be able to steal the master passphrase (and hence the user's files). This script injection may be achieved by hacking the web server, or by tampering with externally loaded scripts on the network, or by exploiting a cross-site scripting (XSS) attack. Many such attacks have been found in previous work [10,7,12] and reports from the MEGA bug bounty program indicate that such attacks are a common concern. Cloud storage websites employ many techniques to block these attack vectors, such as Strict Transport Layer Security [20] and Content Security Policy [24], but the increasing incidence of server-side compromises, man-in-the-middle attacks on TLS, and XSS vulnerabilities on websites, indicates that it would be prudent to try to protect user data even if the website had a malicious script.

Even ignoring malicious scripts, to provide any formal security guarantee for a website security component that runs alongside unknown third party scripts, the component would need to be robust against bugs in these scripts. To give a concrete example, the MEGA website relies on about 70 scripts, and less than 10% of their code is related to cryptography; most of the rest implements the user interface. So the correctness of the cryptographic library and the secrecy of its keys relies on the good behavior of these UI scripts, which are not written by security experts and may be difficult to formally review.

1.2 Password Manager Bookmarklets and Browser Extensions

Password managers (e.g. LastPass) help users manage and remember their passwords (and other sensitive data such as credit card numbers) on various websites. They are often implemented in JavaScript and deployed as a browser extension or bookmarklet that detects the login page of a website, looks up a password database for a matching username and password, and offers to fill it in automatically. If there is no matching password, it may offer to generate a difficult-to-guess password and store it in the database. To synchronize and backup the password database across a user's devices, many password managers implement the host-proof encrypted cloud storage pattern described above.

For example, LastPass users can generate a "Login" bookmarklet and add it to their browser's bookmarks. The bookmarklet contains a JavaScript program embedded with an encryption key for the user's password database. When a user next visits the login page at some website, she may click on the bookmarklet to automatically log in. Clicking on the bookmarklet executes its JavaScript program in the scope of the current page. The program contacts the LastPass

website and retrieves the currently logged-in LastPass user's encrypted password data from the cloud server. It then uses the encryption key embedded in the bookmarklet code to decrypt the password for the current page and fills in the login form. If the browser does not have an active login session with LastPass, the bookmarklet has no effect.

The main threat to the bookmarklet design is that it may be clicked on a malicious website that may then tamper with the JavaScript environment to subvert the bookmarklet's functionality. A typical case is if the user accidentally clicks the bookmarklet on a website that looks like a known trusted site. Or the user has passwords for two different sites stored in her database, and one of them may have been compromised. In these situations, the main goal of the malicious website is to steal the user's password at a different honest website. The bookmarklet tries to prevent such attacks by identifying the website the bookmarklet has been clicked on and only using its embedded secret on trusted websites. However, identifying the host website and protecting the bookmarklet secret are difficult in a tampered JavaScript environment, leading to many attacks [2,10,7,12]. We propose a programming discipline that enables secret-keeping bookmarklets that are robust against tampered environments.

As an alternative to bookmarklets, many password managers also provide a downloadable browser extension that executes a similar JavaScript program, but in a safer, more isolated JavaScript context. Password manager browser extensions are subject to their own threat model [9], not detailed here. In particular, even extensions must protect their secrets from being leaked by bugs in other included JavaScript programs. To give a concrete example, the LastPass extension for the Google Chrome browser has 119 JavaScript files, of which only 5 contain any cryptography, but their security guarantees still must rely on the correctness of these other scripts.

1.3 Single Sign-On and Social Sharing Buttons

Single Sign-On protocols (e.g Facebook's Login button) are widely used by websites that wish to implement authenticated sessions without the hassle of user registration and password management. Another advantage is that the website can leverage their users' social networks to provide a richer experience (e.g. Facebook's Like button). From the user's viewpoint, single sign-on and social sharing buttons offer her a convenient and secure way of accessing and sharing data across different websites, without needing to remember different passwords.

For example, to include the Facebook Login button on a web page, a website W loads a JavaScript library provided by Facebook that displays the button. When a user clicks on the button, the program asks Facebook for the currently logged-in user's access token for the current website W. If the user is logged in and has previously authorized Facebook to provide an access token to W, Facebook returns the access token in a URL. Otherwise, the user is forwarded to a page where she may login and authorize W (or not). The program then gives the access token to the website and also provides functions to access the Facebook API and read or write (authorized elements of) the current user's social profile.

The protocol implemented by Facebook is OAuth 2.0 [19], which also prescribes other message flows for server-side tokens and smartphones. Other popular single sign-on protocols, such as OpenID, SAML, and BrowserID, provide similar message flows that websites may use to obtain access tokens as user-specific credentials.

The main threat to the single sign-on interaction above is that the access token may be stolen by a malicious website and then used to impersonate the user at an honest website, or to read or write the user's profile information on her social network. The OAuth 2.0 flow is particularly vulnerable since access tokens are sent in URLs which may be leaked by Referer headers, or by HTTP redirection, or by various browser and application bugs [8,26,12]. Since the access token is used as a bearer token, and is often not specific to a website, it can be immediately used by the adversary on any website to impersonate the user.

The BrowserID single sign-on protocol seeks to mitigate the effects of token theft by using public key cryptography to authenticate the client[5]. Mozilla's implementation of BrowserID is written fully in JavaScript. The client includes a JavaScript cryptography library that may be included by any site to retrieve and sign tokens on behalf of the user. Even the single sign-on server is written in JavaScript and deployed over `node.js`. The design of BrowserID has been carefully evaluated by formal analysis [17], but to prove the code correct, one must show that all the scripts loaded alongise behave safely. In Mozilla's implementation, the server-side protocol moduls is loaded among 158 other `node.js` modules, and a bug or malicious function in any of these modules could compromise both ther server's and user's private keys.

1.4 Towards Verifiably Secure Web Components

We have discussed three popular categories of JavaScript security components that seek to protect sensitive user data such as files, passwords, and access tokens from malicious websites using various combinations of authentication protocols and cryptography. Each of these components is used in conjunction with a number of other scripts that may modify the JavaScript environment.

Our goal is to write JavaScript security components in a style that their security can be formally proved even if the context is malicious. In particular, we aim for a language-level isolation guarantee for our programs — that their input-output functional behavior cannot be tampered with by the environment. As a corollary, any secrets that are correctly protected by cryptography in our programs cannot be stolen or modified by the adversary. This simple-sounding isolation guarantee would be trivial to obtain in traditional programming languages with sound type systems, such as OCaml and Java. However, the flexibility of JavaScript breaks many guarantees presumed by the programmer and the language must be reined in before we can achieve our goal.

In Section 2, we discuss the peculiarities of JavaScript and the browser environment that make it difficult to isolate security components. In Section 3, we

[5] `http://login.persona.org`

present Defensive JavaScript (DJS), a typed subset of JavaScript that guarantees isolation from the environment. In Section 4, we present a large cryptographic library written in DJS and use it to write and verify simple cryptographic web applications. Section 5 concludes.

2 Secure Messaging in an Untrusted Environment

As a motivating example, we consider how to implement a JavaScript program that sends an authenticated message to a server. Our target web page is hosted on a website W at URL `http://W.com` and it loads three scripts:

```
<html>
 <body>
 <script src="attacker1.js"></script>
 <script src="messaging.js"></script>
 <script src="attacker2.js"></script>
 </body>
</html>
```

The first and third scripts are arbitrary malicious scripts chosen by the attacker. The second script is our program that provides an API to send messages to a server S at the URL `http://S.com`, via the `XMLHttpRequest` asynchronous messaging API provided by the browser. (In some cases, W may be the same site as S.) We assume that the program and S share a secret MACing key k. The program uses this key to attach a MAC to each message sent to S.

The security goal is message authentication: every message received and verified by S must have been sent by our program running at W. In particular, it should not be possible for the attacker scripts to steal the MAC key k and forge messages to S. The above web page scenario may seem too paranoid, but more generally, we want to guarnatee that that even if the surrounding scripts are just buggy, not malicious, they still cannot accidentally leak the key.

2.1 Secure Delivery of the Secret Key

The first challenge is to deliver the MAC key to `messaging.js` in a way that cannot be read by the other two attacker scripts.

Injecting the key as a token into the HTML document, or an HTTP cookie, or in browser local storage would not work; if the messaging script can read it, so can the attacker's script. The only safe place for the key is to embed it into the messaging program. But even in this case, there are many pitfalls. Consider the following messaging script `messaging.js` with a key included on top:

```
var key = k;
var api = function(msg){ .../*send authenticated message*/}
```

Unfortunately, the attacker script `attacker2.js` can simply read the variable `key` from the environment and obtain the key. A better solution would be to protect the key within the function:

```
var api = function(msg){
  var key = k;
  .../*send authenticated message*/
}
```

Now the script `attacker2.js` can no longer read the local variable `key`. However, it can retrieve the source code of the function `api` as a string by calling `api.toSource()`. It can then extract the embedded key k from the string. To protect the source code of the function, we need to rewrite the function by wrapping it within an anonymous function closure:

```
var api = (function (){
            var _api = function(msg){
                        var key = k;
                        .../*send authenticated message*/}
            return function(msg){return _api(msg);}
          )();
```

Now, calling `api.toSource()` only reveals the code of the wrapper function, and the code of the real `_api` function (which embeds the key k) remains private.

There remains another way for the attacker scripts to obtain the source code of `_api`. If the script `messaging.js` is served from the current website's origin `http://W.com`, the source code of the whole script can be retrieved by either attacker script by making an `XMLHttpRequest` to the script's URL:

```
var xhr = new XMLHttpRequest();
xhr.open("GET","http://W.com/messaging.js",false);
xhr.send();
var program = xhr.responseText;
```

To prevent this, the messaging script must be served from a separate origin. For example, the website W could set aside a separate origin for serving only scripts, and place the messaging script at say `http://scripts.W.com/messaging.js`. in our example, it would also be suitable to source it from S's origin, say at `http://S.com/messaging.js`, so S can inject the shared key into the script. In both cases, the attacker scripts on `http://W.com` would be unable to make an `XMLHttpRequest` to read the code, due to the Same Origin Policy.

2.2 Calling External Functions

To construct and send a message, our messaging program will rely on several external functions either builtin to the JavaScript language or provided by the browser as part of the DOM library. For example, commonly used string functions such as concatenation (`s.concat(t)`) or search (`s.indexOf(t)`) are defined as methods in the `String` prototype. Other useful functions on arrays and objects are provided by the `Array` and `Object` prototypes. The `window.Math` object provides implementations of many mathematical functions. The `XMLHttpRequest` object allows asynchronous messaging with remote servers, and the `postMessage` API implements client-side messaging between windows. Finally, the `document`

object (or DOM) provides functions for reading and writing the HTML document (e.g. `document.getElementById(''body'')`).

These external library functions are widely used by JavaScript programs. However, in our threat scenario, the attacker script `attacker1.js` may have redefined every one of these functions by modifying the `String`, `Array`, and `Object` prototypes, or by redefining these functions and objects in the `window` and `document` objects. For example, the following code redefines the `XMLHttpRequest` object, so that all messages send by the messaging script can be intercepted:

```
window.XMLHttpRequest =
 function(){
  return {open: function(){/*do whatever*/},
          send: function(){/*do whatever*/}}}
```

Suppose our messaging program is written as follows; in addition to the `XMLHttpRequest` object (and its methods), the code calls `Crypto.HMAC`:

```
var api = (function (){
            var _api = function(msg){
                        var key = k;
                        var xhr = new XMLHttpRequest();
                        xhr.open("GET","http://S.com",false);
                        xhr.send(Crypto.HMAC(key,msg) + "," + msg);
                       }
            return function(msg){return _api(msg);}}
          )();
```

This code exemplifies three dangers of calling an external function.

First, the call to `Crypto.HMAC` leaks the key, since the attacker may have redefined the function. Consequently, the only safe choice here is to inline the code of the `HMAC` function into the messaging program. The `HMAC` function in turn relies on a hashing function (say `SHA-256`) which would also need to be included within the program. (To see what these functions look like in JavaScript, see our implementation in Appendix A.)

Second, the call to any external function exposes `_api` function to a stack-walking attack. For example, the attacker can redefine `XMLHttpRequest.send` so that when it is called, it reads the source code of its calling function using the `caller` method in the `Function` prototype:

```
stackwalk = function(){var program = stackwalk.caller.toSource();...}
window.XMLHttpRequest =
 function(){
  return {open: stackwalk,
          send: stackwalk}}
```

Adding the above code in `attacker1.js` will set up the environment such that when `_api` calls `xhr.open`, the attacker obtains the source code of `_api` and hence its embedded key. The attack relies on the implementation of the `caller` method, and it it works at least in Firefox at the time of writing. More generally, this kind of stack-walking is a powerful attack vector. Whenever a function `f` is

called, it can access its caller by accessing `f.caller`, and the next level on the call stack by accessing `f.caller.caller`. At each level, it may examine (and even overwrite) the arguments of the function.

Third, if our messaging script ever calls an external function, the attacker may redefine its behavior so that the result of the function is not as expected. For example, `s.concat(t)` may always return a constant string or `Math.pow` may always return `0`. In such cases, the functional integrity of our script has been compromised, and if the results of these functions are used in the `MAC` function, the authentication protocol may be broken even without leaking the secret key.

In summary, any external function calls from a the messaging script may lead to a full compromise of its secrets and its functionality. To be safe, the script must never call functions from within security sensitive functions whose source code or arguments may be secret. Instead, all external function calls should be factored out into a top-level wrapper function that calls a self-contained API:

```
var api = (function (){
            var hmac = function(key,msg){/* inlined HMAC code */}
            var _api = function(msg){
                        var key = k;
                        return (hmac(key,msg) + "," + msg);
                       }
            return function(msg){return _api(msg);}
          )();
var msg_api = function (msg) {
               var mac = api(msg);
               var xhr = new XMLHttpRequest();
               xhr.open("GET","http://S.com",false);
               xhr.send(mac);
             }
```

Here, the external function call to `XMLHttpRequest` is performed outside the sensitive API by a function `msg_api` that has no access to the secret MACing key. Walking the stack to get to `msg_api` does not allow the attacker to steal any secrets or to tamper with the `_api` function.

2.3 Implicit Calls to External Functions

In addition to explicit function calls, many JavaScript constructs implicitly trigger methods defined in various prototypes. Since these prototypes may be modified by the adversary, we must also avoid such implicit calls in defensive code.

The first category of implicit function calls are *coercions*. For example, in the expression `e == e'`, if `e` is an object and `e'` is a number, then the equality will trigger an implicit coercion `e.valueOf` e was an object; rest of paragraph assumes string. This method `valueOf` is defined in the `String` prototype. More generally, comparison between any object and a string or a number may trigger the `valueOf` or `toString` methods in that object's prototype. Hence, by redefining these methods in the `Object` prototype, the attacker can intercept any function that triggers an implicit coercion and mount the attacks described in the previous subsection.

The second category of implicit function calls are *getters* and *setters*. Whenever an object is accessed at an undefined property (e.g. `o.x`), the JavaSript interpreter traverses the prototype hierarchy to see if the property `x` is defined in one of the prototypes that the object is derived from. If, say, none of the prototypes has defined `x`, but the `Object` prototype defines a getter function for `x`, then reading the property `o.x` will trigger this function. Similarly, if the `Object` prototype has a setter function for `x`, writing to `o.x` will call the setter.

By defining getters and setters for specific properties, an attacker script can cause trusted code to trigger an external function if it ever accesses an undefined property. Similarly, if an array or string is every indexed out of bounds, it may trigger a getter or setter in the `Array` prototype. Consequently, in our setting, the messaging program should never access arrays, strings, or objects outside their declared ranges. In particular, the popular JavaScript idiom of first declaring an empty object and then extending it is vulnerable to attack:

```
1 Object.defineProperty(Object.prototype,"a",{set:function(){...}});
2 var x = {};
3 x.a = 1; // triggers malicious setter
4 Object.defineProperty(Array.prototype,"0",{set:function(){...}});
5 var y = [];
6 y[0] = 1; // triggers malicious setter
7 Object.defineProperty(Array.prototype,"1",{get:function(){...}});
8 y[0] = y[1]; // should be undefined, but triggers malicious getter
```

A particular subcase of prototype poisoning is worth mentioning. JavaScript offers a `for...in` loop construct that goes through all the properties of an object. For example `for (i in {x:1})print(i)` is expected to print ‘‘`x`’’ and `for (i in [1])print(i)` is expected to print the single array index `0`. However, if the attacker modifies `Object` and `Array` prototypes to add more properties, those properties will also be printed here. Even checking that each property was defined locally within the object using the `Object.hasOwnProperty` function does not help, since this function could also be modified by the adversary.

2.4 Defensive Programming Idioms

We have discussed many potential attack vectors that a malicious script may employ when trying to subvert an honest JavaScript program running in the same environment. To prevent these attacks, we advocate a defensive programming discipline where programs aim to isolate their security-critical code from the environment by using function closures, by being loaded from a different origin, by refusing to explicitly call external functions, and by carefully preventing the triggering of coercions and prototype lookups. To systematically check our programs for all these isolation conditions, we propose a static type system. Defensiveness is a first step towards formal security guarantees. Once scripts like our messaging program are correctly isolated, we may rely on their context-independent semantics and on the functional integrity of their cryptographic libraries to build automated security verification tools.

Alternative Mitigations. The injunction that the core messaging API must be fully self-contained may seem draconian and one may wonder if there are some cases in which calling external functions is safe. If the goal is only to prevent stack-walking, one may hide the stack by calling all external functions through a recursive wrapper function [25]. However, this requires a source-to-source translation to implement effectively, especially for object methods like `xhr.send`.

Recent versions of JavaScript give programs the ability to freeze objects and mark various properties as unmodifiable and/or unconfigurable (cannot be deleted). It is tempting to suggest that the website W should freeze some objects or that the browser should guarantee that some DOM properties are unforgeable. These objects and properties would then be safe to access. However, the problem with both `Object.freeze` and `Object.defineProperty` is that they need to apply to the top object in the object hierarchy, otherwise it is ineffective. For example, the properties `document.location.href` and `window.location.href` are commonly considered unforgeable since modifying them would take the webpage to a new location. Indeed, most browsers prevent JavaScript from redefining these properties. However, the attacker may directly redefine the `window.document` object (FireFox) or the `window.location` object (Internet Explorer).

Another option is for the website W to run a script first that makes copies of all relevant objects before they have been tampered by the attacker [18]. However, ensuring that a script runs first on a web page is surprisingly tricky [25]. Moreover, this solution does not work in scenarios where the website W itself may be malicious or compromised.

One may also use isolation mechanisms outside JavaScript, such as HTML `iframes` to effectively separate trusted and untrusted code [4]. In this paper, we do not investigate such mechanisms and instead focus only on language-based isolation. We note that the use of `iframes` relies on the semantics of the Same Origin Policy which remains to be fully standardized, let alone formalized [28]. Furthermore, `iframes` may not be available in some JavaScript runtime environments, such as smartphones and server applications. In these environments, defensive programming becomes necessary.

3 Defensive JavaScript

We present a subset of JavaScript that enforces a strong defensive programming discipline. Our language, Defensive JavaScript (DJS), imposes restrictions on JavaScript code both at the syntactic level and through a static type system. The main elements guiding the design of DJS are as follows:

Static Scopes. The variable scoping rules of JavaScript are notoriously difficult to understand. For example, functions may use local variables before they are declared. More worryingly, if a JavaScript program ever accesses a variable that is not in its local scope, this access may trigger a getter or setter in some prototype object. Consequently, we require that all variables in DJS programs be strictly statically scoped. We impose this by restricting the

occurance of variable declarations (`var`) and by enforcing a strong scoping restriction on the bodies of `with` statements.

Static Types for Functions, Objects, and Arrays. To prevent out-of-bound accesses to object properties, function arguments, and array indices, we require that all these objects be statically types. Notably, this means that the objects and arrays are not extensible and the types of variables cannot be changed. Furthermore, dynamic accesses to arrays and strings are only allowed when the index can be guaranteed to fall within bounds.

Coercion-Free Operations. To avoid triggering coercions, we enforce strict types for all unary and binary operators. Comparisons, for example, can only be performed between expressions of the same types.

Disjoint Heaps. To provide full isolation for our programs, we require that no heap references are imported or exported by DJS code. Importing an external object (array, function) is forbidden since accessing any of its properties may trigger malicious code. Exporting an internal object is forbidden because it may expose internal program state (and secrets) to the attacker. Hence, we require that DJS programs can only export scalar (`string -> string`) APIs.

3.1 Syntax

The syntax of DJS depicted in Figure 1 reflects these design constraints. Since DJS is a subset of JavaScript, much of the syntax is standard JavaScript and we refer the reader to the full language specification for more syntactic details [16].

DJS includes the standard JavaScript literals: booleans, numbers, strings, objects, and arrays. In fact, literals are the only way one may construct an object or an array. DJS does not allow object constructors, and extending an existing object or an array is forbidden.

DJS supports several unary ($\triangleright$) and binary ($\diamond$) operators over numbers, strings, and booleans. Since these operators are built into the language and cannot be modified we can use them freely, except that the type system ensures that we do not trigger coercions.

Left-hand-side expressions denote the various ways that objects, strings, and arrays may be accessed in DJS code. Notably, dynamic accessors are severely limited. For example, properties cannot be accessed via the `e[i]` syntax (where `i` may have been dynamically computed). Instead, they must use the static accessor `e.x`. This helps the typechecker ensure (statically) that only explicitly defined properties are accessed (at runtime).

Arrays (and strings) can be accessed only at indexes that can be statically shown to fall within the array (and string) bounds. We allow three kinds of array indexes. The constant index $e[\eta]$ is allowed when η is known to be within the bounds of the array. The integer index $e[e'\&\eta]$ is allowed when η is an integer ($0 \leq \eta < 2^{30}$) and is within the array bounds. The bounded access $x[(e \texttt{>>>} 0)\%x.length]$ is always allowed.

Strings can be accessed with the three array access forms as well as a conditional form that checks that the index is within the length of the string before

$a, b, c ::=$	literals
η	numbers $(0, 1, ...)$xo
σ	strings (‘‘...’’)
`true`, `false`	booleans
$[e_1, \ldots, e_n]$	array literals $(n \geq 0)$
$\{x_1 : e_1, \ldots, x_n : e_n\}$	object literals $(n \geq 0)$
$\triangleright ::= +, -, !, \sim$	unary operators
$\diamond ::=$	binary operators
$+, -, *, /, \%$	arithmetic operators
$\&, \mid, \hat{}$, `<<`, `>>`, `>>>`, `>>>=`	bitwise operators
`&&`, `||`	boolean operators
`==`, `!=`, `>`, `<`, `>=`, `<=`	comparison operators
$l, m, n ::=$	left-hand-side expressions
x, `this`.x	variables
$e.x$	object property
$e[\eta]$	constant array index
$e[e' \& \eta]$	integer index $(0 \leq \eta < 2^{30})$
$x[(e \texttt{>>>} 0) \% x.\texttt{length}]$	bounded array index
$(e \texttt{>>>=} 0) < x.\texttt{length} ? x[e] : \sigma$	conditional string index
$e ::=$	expressions
a	literals
l	left-hand-side expressions
$l = e$	assignment
$\triangleright e$	unary operation
$e \diamond e'$	binary operation
$e_f(e_1, \ldots, e_n)$	function application $(n \geq 0)$
$s ::=$	statement
e	expression
`with` (e) s	scope
`if` (e) s_1 `else` s_2	conditional (`else` optional)
`while` $(e) s$	while loop
$\{s_1; \ldots s_n;\}$	sequential composition $(n \geq 0)$
$f ::=$	function expression
`function` $(x_1, \ldots, x_n)\{$	$(n, m, k \geq 0)$
`var` $y_1 = d_1, \ldots, y_m = d_m;$	
$s_1; \ldots s_k;$ `return` $e;\}$	
$d ::= e \mid f$	defined expression
$p_f ::=$	program (wrapping function f)

```
(function () {
   var _ = f;
   return function (x) {if (typeof x == ''string'') return _(x); }})();
```

Fig. 1. Defensive JavaScript: Syntax

accessing it, otherwise it returns a new string constant. The restrictions on dynamic accesses to objects, strings, and arrays are governed by the limits of our type system and type inference algorithm. With a more expressive type system, one may be able to allow other safe dynamic accessors.

Expressions include assignments, unary and binary operations, and function and method applications. Functions and methods must be fully applied; we do not allow optional arguments that may be left undefined.

Statements include if-then-else conditionals, while loops, and sequencing. Notably, variable declarations `var x` cannot appear in statements and property enumeration via `for-in` is forbidden. General `for` loops are allowed by the typechecker, even though they are not part of the formal syntax.

There are two mechanisms of introducing scope frames in DJS; functions (and methods) and `with`. The statement `with` (e) s takes an object expression e and makes its properties available as local variables to the statement s. To enforce static scoping, we require that all the free unqualified variables of s be properties in e. That is, looking up a free variable in the body of a `with` statement should never require looking beyond the current `with` context.

The syntax of functions is restricted to make it easier to infer their scope frames and return types. The function body begins with a series of variable declarations; in fact, this is the only place where `var` statements appear in DJS programs. The body continues with a series of statements and ends with a single `return` statement. The function is not allowed to invoke `return` anywhere else.

The top-level program p_f is a wrapper around a single function f; it ensures that the argument to the function is a string, calls the function, and returns the result. The wrapping ensures that the source code of the internal function is not leaked to the environment, and the argument typecheck ensures that the program does not accidentally import an external heap reference.

3.2 Typing

The type rules depicted in Figures 2 and 3 enforce the language restrictions described informally above. We write types and typing environments as follows:

Types and Environments

$\tau ::=$	Types
$\quad$ number | boolean | string | `undefined`	Base Types
$\quad \rho$	Object
$\quad [\tau]_n$	Array of length $n \geq 0$
$\quad (\tau_1, \ldots, \tau_n) \to \tau_e$	Function $n \geq 0$
$\quad (\tau_1, \ldots, \tau_n)[\rho] \to \tau_r$	Method on object of type ρ $(n \geq 0)$
$\rho ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	Object Type $(n \geq 0)$
$\kappa ::=$ f | w	Scope frame kind: `function` or `with`
$\Gamma ::= \varepsilon \mid \Gamma, [\rho]_\kappa$	Typing Environment

Types τ include the primitive base types of JavaScript, plus static types for objects, arrays, functions, and methods. Object types ρ look like records; they

declare a fixed set of properties and assign a static type to each property. Unlike JavaScript, DJS array types $[\tau]_n$ require that all elements of the array must have the same type τ and that the array length (n) must be fixed at initialization.

Each function is given a type $(\tau_1, \ldots, \tau_n) \to \tau_r$, which says that the function expects n arguments with the indicated types $\tau_1, \ldots, \tau_n$ and returns a result of type τ_r. Method types look like function types, except that they have an additional implicit argument — the object within which the method resides, denoted by its type ρ. In DJS, methods may only be invoked with the syntax $e.m(x_1, \ldots, x_n)$; it is, for example, forbidden to copy a method into a variable and invoke it without the object prefix.

Typing environments consist of a sequence of scope frames, where each frame looks like an object type ρ: it declares the types for a set of variables local to the frame. Each frame also has a kind annotation that denotes whether the frame was generated by a `with` statement or by a `function` (or `method`).

Most of the typing rules of DJS (Figures 2 and 3) are straightforwars. We give a brief overview, focusing on the more unusual rules:

- The rules for typing literals (Num, String, BoolTrue, BoolFalse, Object, Array) are standard.
- The casting rules (BoolCast, NumCast, StrCast) allow specific conversions between primitive types that do not trigger coercions.
- The type rules for operators (Concat, UnaryOp, ArithmeticOp, ComparisonOp, BooleanOp) ensure that their arguments are already of the required type, so that no coercions will be triggered during their execution.
- The object access rule (Property) ensures that the property is declared within the object's type.
- The three array access rules (ConstantIndex, IntegerIndex, BoundedIndex) ensure that the index is an unsigned integer between 0 (inclusive) and the array length (exclusive). The additional string indexing rule (ConditionalStringIndex) also ensures that the string is accessed at an unsigned integer index within the string.
- Assignment requires the left and right hand sides to have the same type. Formally, there is no subtyping in DJS, even though the DJS typechecker internally infers subtyping constraints.
- The rules for control-flow statements (Sequence, If, While) are standard.
- The rule for `with` e s (With) introduces a new frame of kind `w` into the typing environment and uses this frame to typecheck the statement s. The frame consists of the properties in the object type of the expression e.
- The variable scoping rules (VarLocal, VarFunctionScope) prescribe how to lookup a variable in the typing environment. We first look for a local variable in the current scope frame (VarLocal); if we fail, and if the current scope frame was introduced by a function definition, we look further back into the environment. If the current scope frame was introduced by a `with` statement, we never look further. Hence, a well-typed `with` context in DJS must define all the variables that may be used in its body, it cannot let any variable lookup escape to the surrounding context.

$$\text{Num}\ \frac{}{\vdash \eta : \mathsf{number}} \qquad \text{String}\ \frac{}{\vdash \sigma : \mathsf{string}}$$

$$\text{BoolTrue}\ \frac{}{\vdash \texttt{true} : \mathsf{boolean}} \qquad \text{BoolFalse}\ \frac{}{\vdash \texttt{false} : \mathsf{boolean}}$$

$$\text{BoolCast}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash !e : \mathsf{boolean}} \qquad \text{NumCast}\ \frac{\Gamma \vdash e : \mathsf{string}}{\Gamma \vdash +e : \mathsf{number}}$$

$$\text{StrCast}\ \frac{\Gamma \vdash e : \mathsf{number}}{\Gamma \vdash e + \texttt{""} : \mathsf{string}} \qquad \text{Concat}\ \frac{\Gamma \vdash e_1 : \mathsf{string} \quad \Gamma \vdash e_2 : \mathsf{string}}{\Gamma \vdash e_1 + e_2 : \mathsf{string}}$$

$$\text{UnaryOp}\ \frac{\Gamma \vdash e : \mathsf{number} \quad \triangleright \in \{-, \sim\}}{\Gamma \vdash \triangleright e : \mathsf{number}}$$

$$\text{ArithmeticOp}\ \frac{\Gamma \vdash e_1 : \mathsf{number} \qquad \Gamma \vdash e_2 : \mathsf{number} \qquad \diamond \in \{+, -, *, /, \%, \&, |, \hat{}, \texttt{<<}, \texttt{>>}, \texttt{>>>}, \texttt{>>>=}\}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{number}}$$

$$\text{ComparisonOp}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\mathsf{number}, \mathsf{string}\} \qquad \diamond \in \{==, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{>=}, \texttt{<=}\}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{boolean}}$$

$$\text{BooleanOp}\ \frac{\Gamma \vdash e : \mathsf{boolean} \quad \Gamma \vdash f : \mathsf{boolean} \quad \diamond \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e \diamond f : \mathsf{boolean}}$$

$$\text{Object}\ \frac{\vdash e_i : \tau_i \quad i \in [1..n]}{\vdash \{x_1 : e_1, \ldots, x_n : e_n\} : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}} \qquad \text{Array}\ \frac{\vdash e_i : \tau \quad i \in [1..n]}{\vdash [e_1, \ldots, e_n] : [\tau]_n}$$

$$\text{Property}\ \frac{\Gamma \vdash e : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i} \qquad \text{ConstantIndex}\ \frac{\Gamma \vdash e : [\tau]_m \quad m > \eta \geq 0}{\Gamma \vdash e[\eta] : \tau}$$

$$\text{IntegerIndex}\ \frac{\Gamma \vdash e : [\tau]_m \quad \Gamma \vdash e' : \mathsf{number} \quad 2^{30} \geq m > \eta \geq 0}{\Gamma \vdash e[e' \& \eta] : \tau}$$

$$\text{BoundedArrayIndex}\ \frac{\Gamma \vdash x : [\tau]_n \quad \Gamma \vdash e : \mathsf{number} \quad n > 0}{\Gamma \vdash x[(e\texttt{>>>=}0)\%x.\texttt{length}] : \tau}$$

$$\text{ConditionalStringIndex}\ \frac{\Gamma \vdash x : \mathsf{string} \quad \Gamma \vdash y : \mathsf{number}}{\Gamma \vdash ((y\texttt{>>>=}0) < x.\texttt{length} ? x[y] : \sigma) : \mathsf{string}}$$

Fig. 2. Defensive JavaScript: Typing Rules (Literals and Expressions)

$$\text{Assign}\ \frac{\Gamma \vdash l : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash l = e : \tau} \qquad \text{Sequence}\ \frac{\Gamma \vdash s_i : \texttt{undefined} \quad i \in [1..n]}{\Gamma \vdash \{s_1; \ldots; s_n;\} : \texttt{undefined}}$$

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

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

$$\text{With}\ \frac{\Gamma \vdash e : \rho \quad \Gamma, [\rho]_{\mathsf{w}} \vdash s : \texttt{undefined}}{\Gamma \vdash \texttt{with}(e)\ s : \texttt{undefined}}$$

$$\text{VarLocal}\ \frac{\Phi(x) = \tau}{\Gamma, [\Phi]_{\kappa} \vdash x : \tau} \qquad \text{VarFunctionScope}\ \frac{x \notin dom(\Phi) \quad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_{\mathsf{f}} \vdash x : \tau}$$

$$\text{FunctionCall}\ \frac{\Gamma \vdash f : (\tau_1, \ldots, \tau_n) \to \tau_r \quad \Gamma \vdash e_i : \tau_i \quad i \in [1..n]}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau_r}$$

$$\text{MethodCall}\ \frac{\begin{array}{c}\Gamma \vdash e : \rho = \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \quad \tau_i = (\tau'_1, \ldots, \tau'_m)[\rho] \to \tau_r \\ \Gamma \vdash e_i : \tau'_i \quad i \in [1..m]\end{array}}{\Gamma \vdash e.x_i(e_1, \ldots, e_m) : \tau_r}$$

$$\text{FunctionDef}\ \frac{\begin{array}{c}\rho_k = \{(x_i : \tau_i)_{i \in [1..n]}, (y_j : \mu_j)_{j \in [1..k]}\} \\ \Gamma, [\rho_{k-1}]_{\mathsf{f}} \vdash d_k : \mu_k \quad k \in [1..m] \\ \Gamma, [\rho_m]_{\mathsf{f}} \vdash s : \texttt{undefined} \quad \Gamma, [\rho_m]_{\mathsf{f}} \vdash e_r : \tau_r\end{array}}{\Gamma \vdash \texttt{function}(\tilde{x})\{\texttt{var}\ y_1\texttt{=}d_1, \ldots, y_m\texttt{=}d_m;\ s\ ;\ \texttt{return}\ e_r\} : \tilde{\tau} \to \tau_r}$$

$$\text{MethodDef}\ \frac{\Gamma \vdash \texttt{function}\ (\texttt{this}, \tilde{x})\{\texttt{body}\} : (\rho, \tilde{\tau}) \to \tau_r}{\Gamma \vdash \texttt{function}\ (\tilde{x})\{\texttt{body}\} : \tilde{\tau}[\rho] \to \tau_r}$$

$$\text{Program}\ \frac{\Gamma \vdash f : \mathsf{string} \to \mathsf{string}}{\Gamma \vdash p_f : \mathsf{string} \to \mathsf{string}}$$

Fig. 3. Defensive JavaScript: Typing Rules (Statements and Programs)

- Function and method calls must be fully applied with arguments of the right types. Additionally, a method may only be called with an object of the expected object type ρ.
- Function definitions introduce a sequence of scope frames. The first frame consists of only the argument variables, and is used to typecheck the first variable declaration. Each successive frame adds one local variable and is used to typecheck the next variable definition. After all local variables have been declared, the rest of the function body is typechecked with a frame that consists of all arguments and all local variables.
- The rule for method definitions is similar to function definitions, except that the body is typechecked in a frame that includes an implicit `this` argument that has the object type ρ declared in the method type.
- Programs have the scalar API type `string -> string`. In practice, the DJS typechecker is more general; it allows programs to export an object containing multiple scalar functions.

These typing rules are implemented by the DJS typechecker, which infers types automatically without any annotations. The source code and an online demo of the typechecker is available at `http://defensivejs.com`.

The DJS language and its type system imposes many restrictions on JavaScript programs. In exchange, well-typed DJS programs enjoy strong isolation guarantees. The key functional integrity property is called *independence* [12]:

Definition 1 (Independence). *A program p_f preserves the* independence *of f if any two sequences of calls to the result of p_f with the same sequence of arguments, interleaved with arbitrary JavaScript code, return the same sequence of return values, as long as no call triggered an exception.*

The other key property, called *encapsulation* [12], guarantees that the DJS program's internal heap is isolated from the environment and that any internal secrets can only be leaked though the exported API.

Definition 2 (Encapsulation). *A program p_f encapsulates f over domain $\mathcal{D}$ if no JavaScript program that runs p_f can distinguish between running p_f and running p'_f for an arbitrary function f' without calling the wrapped function returned by p_f. Moreover, for any tuple of values $\tilde{v} \in \mathcal{D}$, the heap resulting from calling $p_f(\tilde{v})$ is equivalent to the heap resulting from calling $f(\tilde{v})$.*

Well-typed DJS programs are guaranteed both these properties [12].

Theorem 1 (Defensiveness). If $\vdash f : \mathsf{string} \to \mathsf{string}$ then the DJS program p_f encapsulates f over strings and preserves its independence.

4 Writing Defensive Cryptographic Applications

We present several case studies illustrating the use of DJS for building secure web components. We begin by describing three libraries, and then describing applications built with these libraries. Code sizes and verification details for these programs are listed in Table 1. All our libraries, applications, and verification tools are available from `http://defensivejs.com`.

Table 1. Defensive JavaScript Libraries and Verified Applications

Program	LOC	Typechecking	ProVerif LOC	ProVerif
Encodings	339	24ms	-	-
DJCL	1425	300ms	-	-
DJSON and JOSE	433	36ms	-	-
Secure RPC	61	7ms	243	12s
Password Manager Bookmarklet	43	42ms	164	21s
Single Sign-On Library	135	42ms	356	43s
Encrypted Storage API	80	31ms	203	25s

4.1 Encoding and Decoding Strings

JavaScript applications often have to convert between different data encodings. Unicode strings are typically encoded in UTF-8. Byte arrays can be stored in integer arrays and converted either to ASCII strings, where each character represents a byte, or encoded in Base64, say for use in URLs.

Typical website JavaScript relies on a variety of libraries to implicitly and explicitly interconvert between strings and byte arrays in various formats (e.g. `window.atob`, `s.charCodeAt(i)`). Since defensive code cannot rely on these libraries, we built our own encoding library that performs these conversions. The library currently supports byte arrays encoded in Hexadecimal, UTF-8, Base64, and ASCII and conversions between these formats.

The main limitation to using our DJS encoding library is performance, since it has no access to native objects and libraries. Since we cannot trust that the attacker has not tampered with efficient library objects like `Int32Array`, we encode all byte arrays as ASCII strings. Instead of relying on the String methods `fromCharCode` and `charCodeAt`, which may be modified by the adversary, we use large tables that map UTF-8 codes to their byte representations. The resulting performance penalty depends on the amount of data being encoded, and on the browser and hardware being used. We measured its impact on several applications (listed below), and surprisingly, even with the cost of encoding and decoding, DJS applications run as fast or faster than comparable JavaScript code. Of course, encoding performance could be vastly improved if the browser could provide access to an untamperable native library, such as the `String` prototype.

4.2 DJCL: Defensive JavaScript Crypto Library

We built a fully-featured JavaScript cryptography library in DJS, by adapting and rewriting well-reputed libraries like SJCL [23] (for symmetric cryptography) and JSBN (for public-key cryptography). Our implementation covers the following primitives: AES on 256 bit keys in CBC and CCM or GCM modes, SHA-1 and SHA-256, HMAC, RSA encryption and signature on keys up to 2048 bits with OAEP, PKCS1, or PSS padding. All our functions operate on byte arrays

encoded as ASCII strings. Appendix A presents a detailed listing of the full code for our HMAC and SHA-256 functions.

Typing guarantees that the input-output behavior of the cryptographic functions cannot be tampered with by a malicious environment. However, this does not mean that our code correctly implements the cryptographic algorithm, or that it does not accidentally leak its secrets either explicitly in a return value or implicitly via a side-channel. Proving the functional correctness of our cryptographic library or its robustness against side-channels remains an open problem.

We evaluated the performance of various DJCL functions using the `jsperf` benchmark engine[6] on Chrome 24, Firefox 18, Safari 6.0 and IE 9. We found that our AES block function, SHA compression functions and RSA exponentiation performed at least as fast as their SJCL and JSBN counterparts, and sometimes even faster. We conclude that defensive coding is well suited for bit-level, self-contained crypto computations, and JavaScript engines find it easy to optimize our non-extensible arrays and objects.

On the other hand, when implementing high-level constructions such as HMAC or CCM encryption that operate on variable-length inputs, we must pay the cost of encoding and decoding data as ASCII strings. Despite this performance penalty, even on mobile devices, DJCL achieves encryption and hashing rates upwards of 150KB/s, which is sufficient for most applications.

To further exercise our cryptographic library, we built an implementation of the upcoming W3C Web Cryptography API standard [15], which is currently being implemented by various browsers and JavaScript libraries as an extension to the `window` object. We implement this API as a set of non-defensive functions that wrap DJCL. We compared the performance of our implementation on benchmarks provided by Chrome and Microsoft; our code is as fast as both native and JavaScript implementations provided by mainstream browsers.

4.3 JSON Serialization

Messaging applications in JavaScript widely use the JSON format, which is considered more compact and easier to use programmatically than XML. JSON defines a JavaScript-like syntax for serializing scalar objects and arrays. For example, the object `{a:"s",b:[0,1]}` is written in JSON notation as the string `'{"a":"s","b":[0,1]}'`.

All modern browsers provide libraries for serializing and deserializing JSON objects. The function `JSON.stringify` takes any JavaScript object and serializes it as a string, typically by ignoring any functions it finds in the object's structure. Conversely, the function `JSON.parse` takes a string and attempts to reconstruct a scalar JavaScript object from the string. DJS programs cannot use the browser's JSON library, since it may have been tampered by the adversary. So we build a defensive JSON library (DJSON) to provide this functionality.

`DJSON.stringify` is conceptually a simple function; it takes an object, enumerates its properties, and writes them out to a string. However, since the attacker

[6] `http://jsperf.org`

may tamper the `Object` and `Array` prototypes, neither the `for...in` loop nor new APIs like `Object.keys` can be trusted to correctly enumerate properties. Consequently, `DJSON.stringify` takes an additional parameter — an object "schema" that describes the type of the JSON object. For example, to serialize the JSON example above, an application would call `DJSON.stringify` as follows:

```
DJSON.stringify({a:"s",b:[0,1]}, // JSON object
               {type:"object", // Schema object
                props: [
                   {name:"a",value:"string"},
                   {name:"b",value:{
                               type:"array",
                               props:[{name:"0",value:"number"},
                                      {name:"1",value:"number"}]}}]})
```

Given such a schema, `DJSON.stringify` ensures that the given object has all the fields and array indices declared in the schema before returning the serialized string; if the object does not match the schema it returns an error.

Implementing `DJSON.parse` is a bit more challenging, since the function needs to create a new object with an arbitrary number of properties. Creating an empty object and adding properties to it would not work, since the attacker may have set up malicious setter functions on the Object prototype. We define a defensive `DJSON.parse` function that requires three parameters — the string to parse, the schema for the expected JSON object, and a pre-allocated object that matches this expected schema. `DJSON.parse` does not create a new object; instead, it fills in this pre-allocated object. To return to our example, to parse the serialized JSON string, the application would first create an object `result` and then call `DJSON.parse` as follows:

```
var result = {a:"",b:[-1,-1]}; // Pre-allocated JSON object
DJSON.parse('{"a":"s","b":[0,1]}', // Serialized JSON string
           {type:"object", // Schema object
            props: [
              {name:"a",value:"string"},
              {name:"b",value:{
                          type:"array",
                          props:[{name:"0",value:"number"},
                                 {name:"1",value:"number"}]}}]},
           result)
```

The schemas used for these two functions are closely related to the expected object types of the JSON objects. Indeed, our typechecker processes these schemas as type annotations and uses them to infer types for code that uses these functions.

Using explicit schemas with fixed object and array lengths imposes an important restriction; our JSON library only works with objects whose sizes are known in advance to the programmer. We have implemented extensions of DJS that use ML-style algebraic constructors (e.g. `cons`, `nil`) to allow extensible objects and arrays. The resulting encoded objects are less efficient than object literals but more flexible since they can represent dynamically-sized objects.

By combining our cryptographic library DJCL with DJSON, we implemented a family of IETF standards collectively called Javascript Object Signing and Encryption (JOSE) [21]. These standards include JSON Web Tokens (JWT), which specifies authenticated JSON messages, and JSON Web Encryption, which specifies encrypted JSON messages. Our defensive JOSE library interoperates with other implementations of these specfications, and we use it to implement various cryptographic messaging protocols, such as Secure RPC (see below).

4.4 Applications

We briefly describe four DJS applications that we built using our libraries. We ran the DJS typechecker to verify their defensiveness. Furthermore, as we shall see in the next subsection, we also verifed their cryptographic security against both network and web attackers by translation to the applied pi calculus.

Secure RPC. Using the JOSE libraries, we programmed a variation of the secure messaging program of Section 2 in DJS. The program consists of a core typechecked API object that embeds a secret shared between the program and a trusted server S.

The API provides two functions: `makeRequest` takes a string and returns a serialized JWT object containing the argument and its HMAC; `processResponse` takes a string, parses it as a JWT object, verifies the HMAC and returns the payload (or an error). A non-defensive function then wraps this API to implement a secure RPC: it calls `makeRequest` to create the request, sends this request via `XMLHttpRequest` (or `postMessage`) to a recipient, waits for a response, calls `processResponse` and returns the result.

The security goal of this RPC application is authentication and correlation for the request and response. The goal relies on the secrecy of the HMAC key and the correct use of the HMAC function. Defensiveness guarantees that the key is not accidentally leaked, but the authentication protocol implemented by the application may still fail to achieve its goals. For example, the application may leak the key in its outgoing message or serialize the message incorrectly before MACing. We will see how to analyze the cryptographic security of the application using the protocol analyzer ProVerif [13].

Password Manager Bookmarklet. We implemented a version of the LastPass password manager bookmarklet in DJS. The bookmarklet embeds a secret HMAC key, and when it is clicked on a website W, it performs a secure RPC with the LastPass website using this key to retrieve the currently logged in LastPass user's username and password for the website W and fill it in.

The security goal of the bookmarklet is to enable LastPass to authenticate that the user clicked the bookmarklet on the hosting website W. In particular, a malicious website W should not be able to steal the secret HMAC key or impersonate another website S, even if the user clicks the bookmarklet at W. At LastPass, the bookmarklet is authenticated by the secret key, whereas the website W is authenticated by the `Origin` header that the browser sends along

with the `XMLHttpRequest` message. While many previous attacks have been found on password manager bookmarklets [2,10], our DJS-based solution is the first that can verifiably protect the bookmarklet and its secrets in this scenario.

Protecting Single Sign-On Tokens. We implemented a version of the Facebook JavaScript library that uses a DJS component to protect the user's Facebook access token from other scripts on the page. The DJS component embeds the access token and provides an API though which scripts on the page can access an authorized subset of the user's Facebook profile. The DJS script uses the access token as a MAC key to avoid leaking it to the environment.

In this design, malicious scripts on the page can access (parts of) the user's Facebook profile as long as the page is open, but do not get direct (long-term, offline) access to the access token, and they lose all access when the page is closed and the DJS script stops executing. In particular, the malicious script can never use the token to impersonate the user at another website.

An Encrypted Storage API. Our final DJS application implements an API for encrypted cloud storage. User files are encrypted at the client (via DJCL) and uploaded to a cloud server. The file encryption keys themselves are stored encrypted in local storage, using a master encryption key derived from a passphrase that is known only to the user. The user enters the passphrase on a protected login page (served from a distinct origin), and the derived key is subsequently embedded into a DJS script on the main storage service website.

By using DJS, we isolate the application code that implements cryptography from the rest of the page. Hence, an XSS attack on the main website cannot steal the file encryption key or the master encryption key. However, it can still read and modify user files as long as the page is open. As such, our proposed DJS API is the first to protect long term secrets on encrypted cloud storage websites from XSS attacks, unlike many previous designs [7].

4.5 Verifying Applications with ProVerif

Well-typed programs in DJS enjoy functional integrity and heap isolation, so the environment can only interact with them through their exported scalar (`string -> string`) APIs. Even if the environment is malicious, it cannot access or interfere with the internal state of the program. This isolation guarantee makes it possible to analyze a DJS program independently of its environment, an immense advantage over traditional JavaScript.

DJS prevents some kinds of accidental leakage of secrets, but it cannot protect a program that leaks secrets through its exported interface. For example, even a well-typed DJS program may foolishly return a secret in the result of a public function. Furthermore, even though the DJCL cryptographic library is defensive, it cannot ensure that the application uses it correctly to achieve its security goals.

Designing application-layer cryptographic protocols is an error prone task (e.g. see the attacks in [10]). We advocate the use of formal protocol analysis tools that can verify that DJS applications meet their goals.

We define a translation from DJS programs to processes in the applied pi calculus. The translation mimics previous formal translations to the applied pi calculus from F# [11] and Java [6]. These previous works prove translation soundness — every attack on the source program is present in its translation. However, we do not prove any soundness result for our translation.

Appendix B provides a detailed listing of the applied pi calculus translation for a simple DJS program that can send and receive authenticated messages.

Each DJS function is translated to a process following Milner's famous "functions as processes" encoding of the lambda calculus into the pi calculus [22]. The translated process waits for arguments on an input channel, computes the function result and sends it back on an output channel.

The DJS programmer may selectively prefix any function name by `_lib.` (thus placing it in the `_lib` object) to indicate that the code of the function should not be translated; instead the function should be treated as a trusted primitive. For example, we label all cryptographic primitives and encoding functions as trusted. Their code is not verified; instead, calls to these functions are translated to calls to symbolic constructors and destructors in the applied pi calculus.

The JavaScript heap and stack frames are modeled by a global private table `heap` that is indexed by unique references (fresh pi calculus names). Each object, array, function, and local variable corresponds to an entry in the table. A function can read and write an entry as long as it knows its reference.

Programs may contain two kinds of security annotations that will be treated specially in the translation. A function may log a security event by calling `_lib.event`. For example, `_lib.event(Send(a,b,x))` may indicate that `a` is going to use a secret key to authenticate a message `x` to `b`. These are translated to events in the applied pi calculus and are then used to specify authentication goals. A function may also label a certain value as secret (`_lib.secret(x)`). This expression is translated as the application of a private constructor, and is used to specify secrecy goals for an application.

The translated applied pi calculus process is composed with the WebSpi library and analyzed for violations of its security goals using the cryptographic protocol analyzer ProVerif. The WebSpi library models web browsers, web servers, and enables a variety of well-known web and network attacks. To model the malicious JavaScript environment, we give the attacker read and write access to the global `heap` table, but only for the entries for which he knows the references. The attacker cannot forge pointers. In addition, the attacker is given control over all public channels and access to the function input and output channels for the API exported by the DJS program. The attacker cannot directly access the processes corresponding to internal functions.

Table 1 reports the ProVerif verification time for a few DJS applications. As depicted in Appendix B, ProVerif may find a counterexample to the security goals, which probably indicates an attack on the source DJS program. If ProVerif verifies the security goals, one gain some confidence in the application, but we caution that there may be other attacks not captured by our WebSpi model. Occasionally, ProVerif may not terminate, typically when the source program

uses loops or recursive functions. In this case, the programmer may need to edit the source program or its translation to help ProVerif reach a conclusion.

5 Conclusions

We presented the design of DJS, a defensive subset of JavaScript that is particularly suited for programming web security components that may execute in malicious environments. DJS is not meant for programming whole websites. It does not allow access to any external libraries and imposes many language restrictions that may feel awkward to a typical JavaScript programmer, but are necessary for security on malicious websites. We have shown that large libraries such as DJCL and various applications can be programmed in DJS, at little cost to performance but great gains in security. We showed how DJS applications can be automatically verified for security using the cryptographic protocol analyzer ProVerif. As future work, we plan to relax some of the restrictions of DJS by relying on frozen and unforgeable objects in the environment, as well as by using more expressive types to capture more safe programs. We also plan to prove a formal soundness result for our translation from DJS to the applied pi calculus.

References

1. Adida, B.: Helios: Web-based open-audit voting. In: USENIX Security Symposium, pp. 335–348 (2008)
2. Adida, B., Barth, A., Jackson, C.: Rootkits for JavaScript environments. In: WOOT (2009)
3. Akhawe, D., Barth, A., Lam, P., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: IEEE CSF 2010, pp. 290–304 (2010)
4. Akhawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: USENIX Security (2012)
5. Arapinis, M., Bursuc, S., Ryan, M.: Privacy supporting cloud computing: ConfiChair, a case study. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 89–108. Springer, Heidelberg (2012)
6. Avalle, M., Pironti, A., Pozza, D., Sisto, R.: JavaSPI: A framework for security protocol implementation. International Journal of Secure Software Engineering 2, 34–48 (2011)
7. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In: Basin, D., Mitchell, J.C. (eds.) POST 2013. LNCS, vol. 7796, pp. 126–146. Springer, Heidelberg (2013)
8. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: CSF, pp. 247–262 (2012)
9. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Network and Distributed System Security Symposium, NDSS (2010)
10. Bhargavan, K., Delignat-Lavaud, A.: Web-based attacks on host-proof encrypted storage. In: WOOT (2012)
11. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. In: CSFW, pp. 139–152 (2006)
12. Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Language-based defenses against untrusted browser origins. In: 22nd USENIX Security Symposium (2013)

13. Blanchet, B.: Automatic verification of correspondences for security protocols. Journal of Computer Security 17(4), 363–434 (2009)
14. Blanchet, B., Smyth, B.: ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, http://www.proverif.inria.fr/manual.pdf
15. Dahl, D., Sleevi, R.: Web Cryptography API. W3C Working Draft (2013)
16. ECMA International: ECMAScript language specification. Stardard ECMA-262, 3rd edn. (1999)
17. Fett, D., Küsters, R., Schmitz, G.: An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In: 35th IEEE Symposium on Security and Privacy (S&P 2014). IEEE Computer Society (2014)
18. Fournet, C., Swamy, N., Chen, J., Dagand, P., Strub, P., Livshits, B.: Fully abstract compilation to JavaScript. In: POPL 2013 (2013)
19. Hardt, D.: The OAuth 2.0 authorization framework. IETF RFC 6749 (2012)
20. Hodges, J., Jackson, C., Barth, A.: HTTP Strict Transport Security (HSTS). IETF RFC 6797 (2012)
21. IETF: JavaScript Object Signing and Encryption, JOSE (2012), http://tools.ietf.org/wg/jose/
22. Milner, R.: Functions as processes. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 167–180. Springer, Heidelberg (1990)
23. Stark, E., Hamburg, M., Boneh, D.: Symmetric cryptography in JavaScript. In: ACSAC, pp. 373–381 (2009)
24. Sterne, B., Barth, A.: Content Security Policy 1.0. W3C Candidate Recommendation (2012)
25. Swamy, N., Fournet, C., Rastogi, A., Bhargavan, K., Chen, J., Strub, P.Y., Bierman, G.M.: Gradual typing embedded securely in javascript. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 425–438 (2014)
26. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE S&P, pp. 365–379. IEEE Computer Society (2012)
27. Woo, T., Lam, S.: A semantic model for authentication protocols. In: IEEE Symposium on Security and Privacy, pp. 178–194 (1993)
28. Zalewski, M.: Browser Security Handbook

A Defensive HMAC-SHA-256 Code

To illustrate the DJS programming style as it is used in cryptographic libraries, we present below the full code for the HMAC and SHA-256 functions implemented in DJCL. The code shown here is accepted by the DJS typechecker and hence does not rely on any external functions. To see the code for other defensive cryptographic functions and applications and to try out variations of these programs against the DJS tpechecker, visit http://defesnsivejs.com.

```
1 /**
2  * A hashing library to include with Defensive Applications
3  */
4 var hashing = (function()
5 {
6
7  return {
```

```
/** SHA-256 hash function.
  * @param {string} msg message to hash, as a hex string
  * @returns {string} hash, as an hex string.
  * @alias hashing.sha256
  */
  sha256: {
    name: 'sha-256',
    identifier: '608648016503040201',
    size: 32,
    block: 64,

    key: [0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
          0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
          0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
          0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
          0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
          0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
          0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
          0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
          0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
          0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
          0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
          0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
          0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
          0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
          0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
          0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2],

    hash: function(s)
    {
     var s = s + '\x80', len = s.length, blocks = len >> 6,
         chunk = len & 63, res = '', p = '',
         i = 0, j = 0, k = 0, l = 0,
         H = [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
              0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19],
         w = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

     while(chunk++ != 56)
     {
      s+="\x00";
      if(chunk == 64){ blocks++; chunk = 0; }
     }

     for(s+="\x00\x00\x00\x00", chunk=3, len=8*(len-1);
         chunk >= 0; chunk--)
      s += encoding.b2a(len >> (8*chunk) &255);

     for(i=0; i < s.length; i++)
     {
      j = (j<<8) + encoding.a2b(s[i]);
```

```
58        if((i&3)==3){ w[(i>>2)&15] = j; j = 0; }
59        if((i&63)==63) this._round(H,w);
60       }
61
62       for(i=0; i < H.length; i++)
63        for(j=3; j >= 0; j--)
64         res += encoding.b2a(H[i] >> (8*j) & 255);
65
66       return res;
67       },
68
69      _round: function(H,w)
70      {
71       var a = H[0], b = H[1], c = H[2], d = H[3], e = H[4],
72           f = H[5], g = H[6], h = H[7], t = 0, u = 0, v = 0, tmp = 0;
73
74       for(t=0; t < 64; t++)
75       {
76        if(t < 16) tmp = w[t&15];
77        else
78        {
79         u = w[(t+1)&15]; v = w[(t+14)&15];
80         tmp = w[t&15] = ((u>>>7 ^ u>>>18 ^ u>>>3 ^ u<<25 ^ u<<14) +
81                         (v>>>17 ^ v>>>19 ^ v>>>10 ^ v<<15 ^ v<<13) +
82                         w[t&15] + w[(t+9)&15]) | 0;
83        }
84
85        tmp = (tmp + h + (e>>>6 ^ e>>>11 ^ e>>>25 ^ e<<26 ^ e<<21 ^ e<<7)
86              + (g ^ e & (f^g)) + this.key[t&63]);
87        h = g; g = f; f = e; e = d + tmp | 0; d = c; c = b; b = a;
88        a = (tmp + ((b&c) ^ (d&(b^c))) + (b>>>2 ^ b>>>13 ^ b>>>22 ^ b<<30
               ^ b<<19 ^ b<<10)) | 0;
89       }
90
91       H[0]=H[0]+a|0; H[1]=H[1]+b|0; H[2]=H[2]+c|0; H[3]=H[3]+d|0;
92       H[4]=H[4]+e|0; H[5]=H[5]+f|0; H[6]=H[6]+g|0; H[7]=H[7]+h|0;
93      }
94    },
95
96 /** The hash function to use for HMAC, hashing.sha256 by default
97   * @alias hashing.hmac_hash
98   */
99    hmac_hash: sha256,
100
101 /** HMAC: Hash-based message authentication code
102   * @param {string} key key of the authentication
103   * @param {string} msg message to authenticate
104   * @returns {string} authentication code, as an hex string.
105   * @alias hashing.HMAC
106   */
```

```
  HMAC: function(key, msg)
  {
   var key = key+'', msg = msg+'', i = 0, h = this.hmac_hash,
       c = 0, p = '', inner = "", outer = "";

   if(key.length > h.block) key = h.hash(key);
   while(key.length < h.block) key += "\x00";

   for(i=0; i < key.length; i++)
   {
    c = encoding.a2b(key[i]);
    inner += encoding.b2a(c ^ 0x36);
    outer += encoding.b2a(c ^ 0x5C);
   }

   return encoding.astr2hstr(h.hash(outer + h.hash(inner + msg)));
  }
 };
})();
```

B Verification Example

Source Program. We begin with the following DJS program that uses a cryptographic hash function `_lib.hmac` (as defined above) to authenticated messages between two scripts that are running on the same malicious page and which share a symmetric HMAC key.

Both scripts run the same core DJS program that embeds the key `mac_key` and provides an API with three functions:

- `mac` takes a string message `x`, logs a security event `Send(x)`, and returns the HMAC of `x` using the key `mac_key`.
- `verify` takes a string message `x` and a string `t` and verifies that `t` is the HMAC of `x` using `mac_key`. It then logs the event `Accept(x,t,res)` with the boolean result of the verification and returns the boolean.
- `guess` is used to specify syntactic secrecy. It takes a string argument `k` and logs the event `Leaked(k,true)` if `k` is the same as the secret key `mac_key`.

These core functions may be used by untrusted wrapper functions to create messages that are then sent from one script to other via any communication mechanism, such as `window.postMessage`. We assume that this external wrapper code is under the control of the adversary, who may subvert it by tampering with the `window` object. Hence, for verification, we assume that the attacker can directly call our core API and state our goals using security events in this API.

The intuition for the security events is that whenever `Accept(x,t,true)` is logged for a message `x`, that is the recipient program accepts `x`, it must be the case that `Send(x)` has been logged before, that is the sender program must have intended to send `x`. This is called a correspondence assertion [27] and is a common

way of formalizing authentication goals in cryptographic protocols. Conversely, we expect that the event `Leaked(k,true)` is never logged, that is the HMAC key remains unknown to the adversary.

These authentication and secrecy queries are embedded on the top of the script using the `_lib.spec` function, which tells the ProVerif translator to directly embed its argument into the generated scripts. We run the ProVerif translator on this simple DJS library and verify that the API satisfies these queries.

```
/* Declaring Events */
_lib.spec("event␣Send(String)");
_lib.spec("event␣Accept(String,String,Boolean)");
_lib.spec("event␣Leaked(String,Boolean)");

/* Sanity Check: Are the Events Reachable? */
_lib.spec("query␣x:String;␣event(Send(x))");
_lib.spec("query␣x:String,t:String;␣event(Accept(x,t,bool_true()))");

/* Authentication Query */
_lib.spec("query␣x:String,t:String;␣event(Accept(x,t,bool_true()))␣==>␣
    event(Send(x))");

/* Secrecy Query */
_lib.spec("query␣x:String;␣event(Leaked(x,bool_true()))");

x = (function()
{
  var mac_key = _lib.secret("xxx");

  var mac = function (x) {
    _lib.event(_lib.Send(x));
    return _lib.hmac(x, mac_key);
  }

  var verify = function (x,t) {
    var res = _lib.hmac(x, mac_key) === t;
    _lib.event(_lib.Accept(x,t,res));
    return res ? "yes" : "no";
  }

  var guess = function (k) {
    var res = k == mac_key;
    _lib.event(_lib.Leaked(k,res));
    return res ? "yes" : "no";
  }

  var _ = function(s)
  {
   var o = _lib.DJSON_parse(s, {t: "", h: ""});
   var h = o.h;

```

```
// oops = mac_key;
  return (o.t == "" ? guess(h) :
         (h == "" ? mac(o.t) :
          verify(o.t, h)));
 }

 return function(s){if(typeof s=="string") return _(s)};
})();
```

Generated Model. The ProVerif script generated by our model extraction tool (DJS2PV) is presented below to illustrate the translation. The script uses the typed applied pi calculus syntax described in [14]. It shows how the various DJS objects and variables are stored in the `heap` table, how the functions are encoded as processes, how the call to the `_lib.hmac` function is turned into a function call, and how all constant strings are extracted as top-level declarations.

The generated script relies on an external WebSpi library that defines all the types (`String`, `Boolean`, `MemLoc`, `Function`), the cryptographic functions (`hmac`, `secret`), and a table representing the JavaScript heap (`heap`). The WebSpi library encodes a rich attacker model that includes both web and network attacks. To verify DJS, we extend WebSpi to allow the attacker direct access to the JavaScript heap: the attacker can insert any object into the heap and read any object for which he knows the table index (representing the heap reference). More details on the original WebSpi library can be obtained from `http://prosecco.inria.fr/webspi`. Other verification examples that rely on WebSpi have appeared in [8,7].

```
free var_x:Memloc.

free str_1:String.
free str_2:String.
free str_3:String.
free str_4:String.
free str_5:String.

event Send(String).
event Accept(String,String,Boolean).
event Leaked(String,Boolean).
query x:String; event(Leaked(x,bool_true())).
query x:String; event(Send(x)).
query x:String,y:String; event(Accept(x,y,bool_true())).
query x:String,y:String; event(Accept(x,y,bool_true())) => event(Send(x))
    .

process
(new fun_1:channel;
(!in(fun_1, (ret_1:channel));
new var_mac_key:Memloc;
insert heap(var_mac_key,mem_string(secret(str_1)));
new var_mac:Memloc;
```

```
new fun_2:channel;
(!in(fun_2, (ret_2:channel,arg_x:String));
new var_x:Memloc;
insert heap(var_x,mem_string(arg_x));
get heap(=var_x, mem_string(val_1)) in
event Send(val_1);
get heap(=var_x, mem_string(val_2)) in
get heap(=var_mac_key, mem_string(val_3)) in
out(ret_2,hmac(val_2,val_3));
0)|
insert heap(var_mac,
  mem_function(function(fun_2)));
new var_verify:Memloc;
new fun_3:channel;
(!in(fun_3, (ret_3:channel,arg_x:String,arg_t:String));
new var_x:Memloc;
insert heap(var_x,mem_string(arg_x));
new var_t:Memloc;
insert heap(var_t,mem_string(arg_t));
new var_res:Memloc;
get heap(=var_x, mem_string(val_4)) in
get heap(=var_mac_key, mem_string(val_5)) in
get heap(=var_t, mem_string(val_6)) in
insert heap(var_res,mem_boolean(equal(mem_string(hmac(val_4,val_5)),
    mem_string(val_6))));
get heap(=var_x, mem_string(val_7)) in
get heap(=var_t, mem_string(val_8)) in
get heap(=var_res, mem_boolean(val_9)) in
event Accept(val_7,val_8,val_9);
get heap(=var_res, mem_boolean(val_10)) in
let val_11=(if val_10=bool_true() then str_2 else str_3) in
out(ret_3,val_11);
0)|
insert heap(var_verify,
  mem_function(function(fun_3)));
new var_guess:Memloc;
new fun_4:channel;
(!in(fun_4, (ret_4:channel,arg_k:String));
new var_k:Memloc;
insert heap(var_k,mem_string(arg_k));
new var_res:Memloc;
get heap(=var_k, mem_string(val_12)) in
get heap(=var_mac_key, mem_string(val_13)) in
insert heap(var_res,mem_boolean(equal(mem_string(val_12),mem_string(
    val_13))));
get heap(=var_k, mem_string(val_14)) in
get heap(=var_res, mem_boolean(val_15)) in
event Leaked(val_14,val_15);
get heap(=var_res, mem_boolean(val_16)) in
let val_17=(if val_16=bool_true() then str_2 else str_3) in
```

```
out(ret_4,val_17);
0)|
insert heap(var_guess,
  mem_function(function(fun_4)));
new var__:Memloc;
new fun_5:channel;
(!in(fun_5, (ret_5:channel,arg_s:String));
new var_s:Memloc;
insert heap(var_s,mem_string(arg_s));
new var_o:Memloc;
get heap(=var_s, mem_string(val_18)) in
insert heap(var_o,mem_object(DJSON_parse(val_18,obj_add(obj_add(obj_empty
    (), obj_prop(str_4, mem_string(string_empty))), obj_prop(str_5,
    mem_string(string_empty))))));
new var_h:Memloc;
get heap(=var_o, mem_object(val_19)) in
insert heap(var_h,mem_string(obj_property_string(val_19,str_4)));
get heap(=var_o, mem_object(val_32)) in
get heap(=var_guess, mem_function(val_20)) in
get heap(=var_h, mem_string(val_21)) in
let function(fun_6)=val_20 in
new ret_6:channel;
out(fun_6, (ret_6,val_21));
in(ret_6, val_22:String);
get heap(=var_h, mem_string(val_30)) in
get heap(=var_mac, mem_function(val_23)) in
get heap(=var_o, mem_object(val_24)) in
let function(fun_7)=val_23 in
new ret_7:channel;
out(fun_7, (ret_7,obj_property_string(val_24,str_5)));
in(ret_7, val_25:String);
get heap(=var_verify, mem_function(val_26)) in
get heap(=var_o, mem_object(val_27)) in
get heap(=var_h, mem_string(val_28)) in
let function(fun_8)=val_26 in
new ret_8:channel;
out(fun_8, (ret_8,obj_property_string(val_27,str_5),val_28));
in(ret_8, val_29:String);let val_31=(if equal(mem_string(val_30),
    mem_string(string_empty))=bool_true() then val_25 else val_29) in
let val_33=(if equal(mem_string(obj_property_string(val_32,str_5)),
    mem_string(string_empty))=bool_true() then val_22 else val_31) in
out(ret_5,val_33);
0)|
insert heap(var__,
  mem_function(function(fun_5)));
new fun_9:channel;(!in(fun_9, (ret_9:channel, arg_s:Memval));let
    mem_string(s)=arg_s in (new var_s:Memloc;insert heap(var_s,arg_s);
    get heap(=var__, mem_function(val_34)) in
get heap(=var_s, mem_string(val_35)) in
let function(fun_10)=val_34 in
```

```
115 new ret_10:channel;
116 out(fun_10, (ret_10,val_35));
117 in(ret_10, val_36:String);
118 out(ret_9,val_36);
119 0) else out(ret_9, undefined()))|
120 out(ret_1,function(fun_9));
121 0)| let function(fun_11)=function(fun_1) in
122 new ret_11:channel;
123 out(fun_11, (ret_11));
124 in(ret_11, val_37:Function);
125 insert heap(var_x, mem_function(val_37));
126 0) |
127 attackerHeap()
```

Example Attack. If line 42 is uncommented in the source DJS program (causing the key to be accidentally written to a global variable `oops`, ProVerif is able to show that the `Leaked` event is triggered, and produces the following trace. (Note: this bug is also caught by the DJS typechecker as a defensiveness violation.)

```
1 new fun_8 creating fun_398886 at {1}
2 new ret_260 creating ret_398877 at {419}
3 out(fun_398886, ret_398877) at {420} received at {3} in copy a_398865
4 new var_mac_key creating var_mac_key_398884 at {4} in copy a_398865
5 new k_11 creating k_398878 at {6} in copy a_398865
6 insert heap(var_mac_key_398884,mem_string(k_398878)) at {7} in copy
    a_398865
7 new var_mac creating var_mac_399613 at {8} in copy a_398865
8 new fun_12 creating fun_399614 at {9} in copy a_398865
9 insert heap(var_mac_399613,mem_function(function(fun_399614))) at {19} in
     copy a_398865
10 new var_verify creating var_verify_399615 at {20} in copy a_398865
11 new fun_18 creating fun_399616 at {21} in copy a_398865
12 insert heap(var_verify_399615,mem_function(function(fun_399616))) at {43}
     in copy a_398865
13 new var_guess creating var_guess_398890 at {44} in copy a_398865
14 new fun_31 creating fun_398879 at {45} in copy a_398865
15 insert heap(var_guess_398890,mem_function(function(fun_398879))) at {63}
    in copy a_398865
16 new var__ creating var___398897 at {64} in copy a_398865
17 new fun_41 creating fun_398880 at {65} in copy a_398865
18 insert heap(var___398897,mem_function(function(fun_398880))) at {402} in
    copy a_398865
19 new fun_249 creating fun_398892 at {403} in copy a_398865
20 out(ret_398877, function(fun_398892)) at {417} in copy a_398865 received
    at {421}
21 insert heap(var_x,mem_function(function(fun_398892))) at {422}
22 in(pub, var_x) at {424} in copy a_398875
23 get heap(var_x,mem_function(function(fun_398892))) at {425} in copy
    a_398875
24 out(pub, mem_function(function(fun_398892))) at {426} in copy a_398875
```

```
25 in(fun_398892, (a_398872,mem_string(DJSON_stringify(a_398871)))) at {405}
       in copy a_398865, a_398873
26 new var_s_253 creating var_s_398896 at {407} in copy a_398865, a_398873
27 insert heap(var_s_398896,mem_string(DJSON_stringify(a_398871))) at {408}
     in copy a_398865, a_398873
28 get heap(var___398897,mem_function(function(fun_398880))) at {409} in
     copy a_398865, a_398873
29 get heap(var_s_398896,mem_string(DJSON_stringify(a_398871))) at {410} in
     copy a_398865, a_398873
30 new ret_257 creating ret_398893 at {412} in copy a_398865, a_398873
31 out(fun_398880, (ret_398893,DJSON_stringify(a_398871))) at {413} in copy
     a_398865, a_398873 received at {67} in copy a_398865, a_398874
32 new var_s creating var_s_398895 at {68} in copy a_398865, a_398874
33 insert heap(var_s_398895,mem_string(DJSON_stringify(a_398871))) at {69}
     in copy a_398865, a_398874
34 new var_o creating var_o_398894 at {70} in copy a_398865, a_398874
35 get heap(var_s_398895,mem_string(DJSON_stringify(a_398871))) at {71} in
     copy a_398865, a_398874
36 insert heap(var_o_398894,mem_object(DJSON_parse(DJSON_stringify(a_398871)
     ,obj_add(obj_add(obj_empty,obj_prop(str_4,mem_string(string_empty)))
     ,obj_prop(str_5,mem_string(string_empty))))))) at {72} in copy
     a_398865, a_398874
37 new var_h creating var_h_400490 at {73} in copy a_398865, a_398874
38 get heap(var_o_398894,mem_object(a_398871)) at {74} in copy a_398865,
     a_398874
39 insert heap(var_h_400490,mem_string(undefined_string)) at {240} in copy
     a_398865, a_398874
40 get heap(var_mac_key_398884,mem_string(k_398878)) at {241} in copy
     a_398865, a_398874
41 insert heap(var_oops,mem_string(k_398878)) at {242} in copy a_398865,
     a_398874
42 in(pub, var_oops) at {424} in copy a_398876
43 get heap(var_oops,mem_string(k_398878)) at {425} in copy a_398876
44 out(pub, mem_string(k_398878)) at {426} in copy a_398876
45 in(fun_398892, (a_398867,mem_string(DJSON_stringify(obj_add(a_398866,
     obj_prop(str_4,mem_string(k_398878)))))) at {405} in copy a_398865,
      a_398868
46 new var_s_253 creating var_s_398891 at {407} in copy a_398865, a_398868
47 insert heap(var_s_398891,mem_string(DJSON_stringify(obj_add(a_398866,
     obj_prop(str_4,mem_string(k_398878))))) at {408} in copy a_398865,
     a_398868
48 get heap(var___398897,mem_function(function(fun_398880))) at {409} in
     copy a_398865, a_398868
49 get heap(var_s_398891,mem_string(DJSON_stringify(obj_add(a_398866,
     obj_prop(str_4,mem_string(k_398878))))) at {410} in copy a_398865,
     a_398868
50 new ret_257 creating ret_398881 at {412} in copy a_398865, a_398868
51 out(fun_398880, (ret_398881,DJSON_stringify(obj_add(a_398866,obj_prop(
     str_4,mem_string(k_398878))))) at {413} in copy a_398865, a_398868
     received at {67} in copy a_398865, a_398869
```

```
52 new var_s creating var_s_398889 at {68} in copy a_398865, a_398869
53 insert heap(var_s_398889,mem_string(DJSON_stringify(obj_add(a_398866,
       obj_prop(str_4,mem_string(k_398878))))) at {69} in copy a_398865,
       a_398869
54 new var_o creating var_o_398888 at {70} in copy a_398865, a_398869
55 get heap(var_s_398889,mem_string(DJSON_stringify(obj_add(a_398866,
       obj_prop(str_4,mem_string(k_398878))))) at {71} in copy a_398865,
       a_398869
56 insert heap(var_o_398888,mem_object(DJSON_parse(DJSON_stringify(obj_add(
       a_398866,obj_prop(str_4,mem_string(k_398878)))),obj_add(obj_add(
       obj_empty,obj_prop(str_4,mem_string(string_empty))),obj_prop(str_5,
       mem_string(string_empty))))) at {72} in copy a_398865, a_398869
57 new var_h creating var_h_398887 at {73} in copy a_398865, a_398869
58 get heap(var_o_398888,mem_object(obj_add(a_398866,obj_prop(str_4,
       mem_string(k_398878))))) at {74} in copy a_398865, a_398869
59 insert heap(var_h_398887,mem_string(k_398878)) at {78} in copy a_398865,
       a_398869
60 get heap(var_mac_key_398884,mem_string(k_398878)) at {79} in copy
       a_398865, a_398869
61 insert heap(var_oops,mem_string(k_398878)) at {80} in copy a_398865,
       a_398869
62 get heap(var_o_398888,mem_object(obj_add(a_398866,obj_prop(str_4,
       mem_string(k_398878))))) at {81} in copy a_398865, a_398869
63 get heap(var_guess_398890,mem_function(function(fun_398879))) at {82} in
       copy a_398865, a_398869
64 get heap(var_h_398887,mem_string(k_398878)) at {83} in copy a_398865,
       a_398869
65 new ret_52 creating ret_398882 at {85} in copy a_398865, a_398869
66 out(fun_398879, (ret_398882,k_398878)) at {86} in copy a_398865, a_398869
        received at {47} in copy a_398865, a_398870
67 new var_k creating var_k_398885 at {48} in copy a_398865, a_398870
68 insert heap(var_k_398885,mem_string(k_398878)) at {49} in copy a_398865,
       a_398870
69 new var_res_33 creating var_res_398883 at {50} in copy a_398865, a_398870
70 get heap(var_k_398885,mem_string(k_398878)) at {51} in copy a_398865,
       a_398870
71 get heap(var_mac_key_398884,mem_string(k_398878)) at {52} in copy
       a_398865, a_398870
72 insert heap(var_res_398883,mem_boolean(equal(mem_string(k_398878),
       mem_string(k_398878)))) at {53} in copy a_398865, a_398870
73 get heap(var_k_398885,mem_string(k_398878)) at {54} in copy a_398865,
       a_398870
74 get heap(var_res_398883,mem_boolean(bool_true)) at {55} in copy a_398865,
        a_398870
75 event(Leaked(k_398878,bool_true)) at {56} in copy a_398865, a_398870
76 (*
77 The event Leaked(k_398878,bool_true) is executed.
78 A trace has been found.
79 RESULT not event(Leaked(x_338058,bool_true)) is false.
80 *)
```