

JavaScript in JavaScript (js.js): Sandboxing third-party scripts

Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta
Princeton University

Abstract

Running on billions of today’s computing devices, JavaScript has become a ubiquitous platform for deploying web applications. Unfortunately, an application developer who wishes to include a third-party script must enter into an implicit trust relationship with the third-party—granting it unmediated access to its entire application content.

In this paper, we present `js.js`, a JavaScript interpreter (which runs in JavaScript) that allows an application to execute a third-party script inside a completely isolated, sandboxed environment. An application can, at runtime, create and interact with the objects, properties, and methods available from within the sandboxed environment, giving it complete control over the third-party script. `js.js` supports the full range of the JavaScript language, is compatible with major browsers, and is resilient to attacks from malicious scripts.

We conduct a performance evaluation quantifying the overhead of using `js.js` and present an example of using `js.js` to execute Twitter’s Tweet Button API.

1 Introduction

The web has undoubtedly become one of the most dominant application deployment platforms. Thanks to its wide support from today’s consumer devices—from desktops and laptops to tablets and smartphones—the web’s scripting language, JavaScript, is available on billions of devices.

One problem facing a web application developer is the implicit trust of including third-party scripts. A third-party script is a JavaScript file included from a party other than the application owner. Examples of commonly used third-party scripts are Google Analytics, Facebook’s Like button, Twitter’s Tweet buttons, and advertising platforms. When including one of these third-party scripts, the application is trusting the third-party

script to execute only what is expected of it. The third-party, however, has full access to the application. It could redirect the page, modify the DOM, or insert malware.

An application owner could download the third-party script and serve it from his or her own servers. This at least ensures that the script being run by the application hasn’t been modified without the application owner knowing. However, this is not always possible with dynamically-generated scripts (*e.g.*, advertisements), and it still doesn’t ensure that the third-party script is not malicious. Third-party services often compress their code (*e.g.*, using the closure compiler), producing a large soup of JavaScript that can make it very difficult for a human or static analyzer to verify its behavior. Alternatively, the application could include the third-party scripts in an `iframe`, but `iframes` still have privileges (*e.g.*, alerts, redirection, etc.) that the application might want to disallow. It also requires cumbersome inter-`iframe` messaging for communication.

Static analyzers can be used to rewrite third-party JavaScript [16, 9, 2, 17] before it gets executed. This is often used on small, user-submitted widgets to guarantee their safety, but doesn’t provide flexible, fine-grained control over the third-party script’s privileges, and therefore, are not applicable to large, third-party libraries. Other approaches extend the browser itself to provide security mechanisms for third-party scripts [12]. While a nice approach, adopting a new standard in all major browsers is difficult and breaks backwards-compatibility.

In this paper, we present `js.js`, a JavaScript interpreter that runs on top of JavaScript. It allows site operators to mediate access to a page’s internals by executing a third-party script inside a secure sandbox. We created a prototype `js.js` implementation by compiling the SpiderMonkey [3] JavaScript engine to LLVM [11] and then translating it to JavaScript using Emscripten [20]. The implementation is used to demonstrate the `js.js` API security features and benchmark performance on the SunSpider benchmark suite [4].

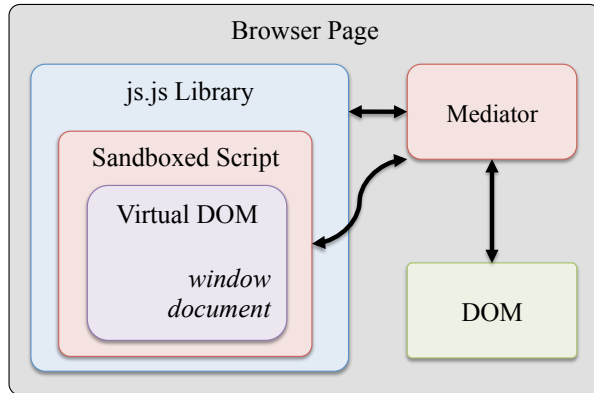


Figure 1: js.js architecture for example application

2 Design

The design goals for js.js are as follows:

- **Fine-grained control:** Rather than coarse-grained control, *e.g.*, disallowing all DOM access, an application should have fine-grained control over what actions a third-party script can perform.
- **Full JavaScript Support:** The full JavaScript language should be supported, including *with* and *eval*, which are impossible to support with static analysis.
- **Browser Compatibility:** All major browsers should be supported without plugins or modifications.
- **Resilient to attacks:** Resilient to possible attacks such as page redirection, spin loops, and memory exhaustion.

With these goals in mind, the js.js API has been designed to be very generic, similar in structure to the SpiderMonkey API. Rather than being specific to a web environment, the js.js API can be used to bind any kind of global object inside the sandbox space. Initially, a sandboxed script has no access to any global variables except for JavaScript built-in types (*e.g.*, Array, Date, and String), but the application can add additional names. In the web environment, these include global names like *window* and *document*. The js.js API allows an application, for example, to add a global name called *alert* that, when called inside the sandbox, calls a native JavaScript function. This way, the application using js.js has complete control over the sandboxed script since the only access the sandbox has to the outside is through these user defined methods. Thus these methods must give the script access only to the elements that the user allows.

Figure 1 shows an example application architecture using js.js. The *Mediator* is a JavaScript application that uses the js.js library to execute a third-party script in a sandbox. The *Virtual DOM* is comprised of the usual web-specific global variables that a script expects, but instead of referring directly to the browser, the media-

```
var src = "nativeAdd(17, 2.4);";
var jsObjs = JSJS.Init();

function nativeAdd(d1, d2) {
    return d1 + d2;
}

var dblType = JSJS.Types.double;
var wrappedNativeFunc = JSJS.wrapFunction({
    func: nativeAdd,
    args: [dblType, dblType],
    returns: dblType});

JSJS.DefineFunction(jsObjs.cx, jsObjs.glob,
    "nativeAdd", wrappedNativeFunc, 2, 0);

var rval = JSJS.EvaluateScript(jsObjs.cx,
    jsObjs.glob, src);

//Convert result to native value
var d = rval && JSJS.ValueToNumber(jsObjs.cx
    , rval);
```

Figure 2: Example of binding a native function to the global object space of a sandboxed script.

tor intercepts all access, such that it can allow or reject requests.

The js.js API aims to be easy to use and flexible. The example in Figure 2 demonstrates using the API to bind to the sandboxed environment, a global function called *nativeAdd* that accepts two numbers as arguments and returns the sum. *Init* initializes a sandboxed environment with standard JavaScript classes and an empty global object. *wrapFunction* is a helper function that allows an application to specify expected types of a function call. If the wrong types are passed to the function, an error is triggered in the sandbox space, which will result in an error handler being called in native space, allowing applications to detect sandboxed errors. *DefineFunction* binds the wrapped function to a name in the global object space of the sandbox, *EvaluateScript* executes the script, and *ValueToNumber* converts the result of evaluating the expression to a native number.

In addition to primitive types like bool, int, and double, the js.js API also includes helper functions for binding more complex types like objects, arrays, and functions to the sandboxed space. With the js.js API, an application can expose whatever functionality of the DOM it wants to a sandboxed script. It can also be used to run user-submitted scripts in a secure way, even providing a custom application-specific API to its users. Currently, creating this virtualized DOM is fairly complex. As future work, we wish to extend the js.js API to make it easier to use by allowing the user to easily setup white/black lists of browser elements, sites, etc.

	Fine-Grained DOM Control	Full JS Support	Page Redirection	Spin Loop / Terminate	Memory Exhaustion	Suspend / Resume
Direct Include	✗	✓	✗	✗	✗	✗
iframe	✗	✓	✗	✗	✗	✗
Web Worker	✗	✓	✓	✓	✗	✗
Static	✓	✗	✓	✗	✗	✗
Static + Runtime	✓	✗	✓	✗	✗	✗
js.js	✓	✓	✓	✓	✓	✓

Figure 3: Table of related work and the attack vectors which they can protect against or support (✓) and those that they cannot (✗). “Static” refers to purely static techniques such as ADsafe and Gatekeeper, while “Static + Runtime” refers to techniques such as Caja and WebSandbox.

The power of *eval* and *with* make them difficult to execute securely. For example, malicious code can use *eval* to disable or circumvent any protections that have been added through JavaScript code. Thus, most other techniques either completely prohibit using them or provide some limited version. However, the powerful sandboxing of scripts that js.js employs means that even *eval* and *with* can be executed securely as they can still only access the protected Virtual DOM.

Since js.js contains a full JavaScript interpreter (a compiled version of SpiderMonkey in our prototype implementation), it supports all variants of JavaScript that the interpreter supports. The js.js API allows an application to specify what version of the JavaScript language to use, anywhere from 1.0 to 1.8. Since an application can bind any name to the sandboxed space, full DOM support can be emulated. In addition, js.js is currently compatible with Google Chrome 7+, Firefox 4+, and Safari 5.1+. Because it requires Typed Array support, js.js will not support Internet Explorer until version 10 (currently in development) is released.

Another benefit of having the interpreter in JavaScript is that js.js has full execution and environmental control of sandboxed scripts. Thus it is relatively simple for js.js to prevent scripts staying in infinite loops or consuming large quantities of memory by placing optional checks inside the interpreter loop. This kind of protection is typically not possible in a normal protection system given the nature of JavaScript.

As seen in Figure 3, js.js is the only technique to meet all of our desired goals. A more detailed discussion of related work can be found in Section 6.

3 Implementation

Our initial prototype implementation of the js.js runtime has been created by compiling the SpiderMonkey [3] JavaScript interpreter to LLVM [11] bytecode using the Clang compiler and then using Emscripten [20] to trans-

late the LLVM bytecode to JavaScript.

Emscripten works by translating each LLVM instruction into a line of JavaScript. Typed Arrays (a browser standard) allows Emscripten to emulate the native stack and heap, such that loads and stores can be translated to simple array accesses. When possible, Emscripten translates operations to native JavaScript operations. For example, an add operation is translated into a JavaScript add operation. An LLVM function call is translated into a JavaScript function call. It also has its own version of a libc implementation. By doing this, the translated output can achieve good performance.

SpiderMonkey comprises about 300,000 lines of C and C++. A lot of our implementation effort was spent patching SpiderMonkey so that it compiles into LLVM bytecode that is compatible with Emscripten’s translator. Due to JavaScript’s inability to execute inline assembly, the JIT capabilities of SpiderMonkey were disabled. We also contributed patches to Emscripten for corner cases that it had previously not encountered. We then wrote the js.js API wrapper around the resulting interpreter. Thus the js.js API greatly resembles SpiderMonkey’s JSAPI.

The translated SpiderMonkey shared library (`libjs.js`) is 365,000 lines of JavaScript and 14MB in size. After closure compiling (`libjs.min.js`), it is 6900 lines and 3MB in size. After gzipping (which all browsers support), it is 594KB. Our wrapper API is about 1000 lines of code. The compiled SpiderMonkey library is available under the Mozilla Public License, while the rest (wrapper script and build scripts) are available under a BSD License. The library can be found at <https://github.com/jterrace/js.js>.

4 Demo Application

To give an example of running a third-party script using js.js, this section describes how to run Twitter’s Tweet Button inside js.js. Twitter makes a script available for embedding a button on an application’s website. Nor-

mally, an application loads Twitter’s script directly from `platform.twitter.com/widgets.js`. Instead, we serve the *unmodified* widget code from our own server, as this version is confirmed to work with `js.js`, and use `js.js` to interpret it.

The Twitter widget script is 47KB of complicated, closure-compiled JavaScript. It expects to be running inside a web page with full access to the DOM. The basic flow of the script is as follows: a large amount of boilerplate code runs first that checks browser compatibility, the DOM is searched for `<a>` elements that match the twitter class selector, and each match is replaced with a new `<iframe>` element containing the Twitter button. Given that it spans a large portion of the DOM API, it is a good representative example of third-party scripts. Supporting this in `js.js` involved all of the following functionality:

- Binding many global objects to the sandbox space that the browser compatibility code checks for, such as `location`, `screen`, `navigator`, and `window` along with many of their properties and functions.
- Allowing the sandboxed code to bind to event handlers such as `DOMContentLoaded`, the `<iframe>` `onload` event, and the `message` event handler (used for inter-iframe communication).
- Many document utility functions such as `getElementsByTagName`, `getElementById`, and `createElement`.
- Wrapping real DOM elements with sandbox-space objects that provide functions like `getAttribute` and `setAttribute`, returning the real DOM element attributes when necessary.

When providing these objects, methods, functions, and handlers, we only provided the sandboxed code with just enough functionality that it can achieve its goal—creating a Twitter button—without allowing it access to any other unnecessary functionality. This demo script can be found at <http://jterrace.github.com/js.js/twitter/>.

5 Evaluation

We evaluate the `js.js` prototype implementation with both microbenchmarks of its API functions as well as with the SunSpider JavaScript Benchmark Suite [4]. The evaluation platform is a Macbook Pro with a 2.4 GHz P8600 and 4GB of RAM. The native tests were performed using SpiderMonkey (tag 20111220) with the JIT disabled. The `js.js` runtime was compiled from SpiderMonkey (tag 20110927) using clang and LLVM version 3.0, and Emscripten version 2.0.

Figure 4 shows the mean time (across ten executions) required to execute the startup and shutdown routines for the `js.js` runtime as well as the time required to evaluate

Function	Time (ms)
libjs.min.js load	84.9
NewRuntime	25.2
NewContext	35.8
GlobalClassInit	15.5
StandardClassesInit	60.1
Execute 1+1	70.6
DestroyContext	33.3
DestroyRuntime	1.8

Figure 4: Mean (across 10 executions) runtime for various `js.js` initialization and execution procedures.

a simple `1+1` expression. The overhead of creating the runtime environment to start executing a script is not an expensive cost.

Figure 5 shows the SunSpider benchmark results for `js.js` in both Chrome and Firefox. For Chrome, most benchmarks fall in the 100x to 200x slowdown range, while Firefox lags behind in some benchmarks. This slowdown is due to a combination of inefficiency in the Emscripten compiler and the overhead of running JavaScript within JavaScript. Further efforts to improve the Emscripten compiler, along with manual optimization of the resulting JavaScript, could result in even better performance. However, this level of overhead could be acceptable to protect websites from untrusted third-party scripts. Trusted JavaScript code can still run natively alongside `js.js` protected code, especially effective if `js.js` is running in a Web Worker¹. The majority of the execution time is spent in `js.js`’s main interpreter loop, a very large function that Emscriptenm, and JIT compilers in browsers, do not do very well at optimizing. We are currently working on ways to improve the performance of this function.

Also note that although the performance overhead of running `js.js` is high, other implementations of the `js.js` API could improve performance. For example, an implementation of the API could be built with Native Client [6] or incorporated directly into future versions of browsers, with the pure JavaScript implementation being used as a fallback if no faster implementations are available.

6 Related Work

iframes have been widely adopted because of the flexibility they provide for sandboxing third-party pages. An iframe allows each third-party script to have its own page

¹Note that since the DOM API is synchronous and Web Workers are asynchronous, a blocking mechanism, such as the HTML5 file-system API, would have to be used.

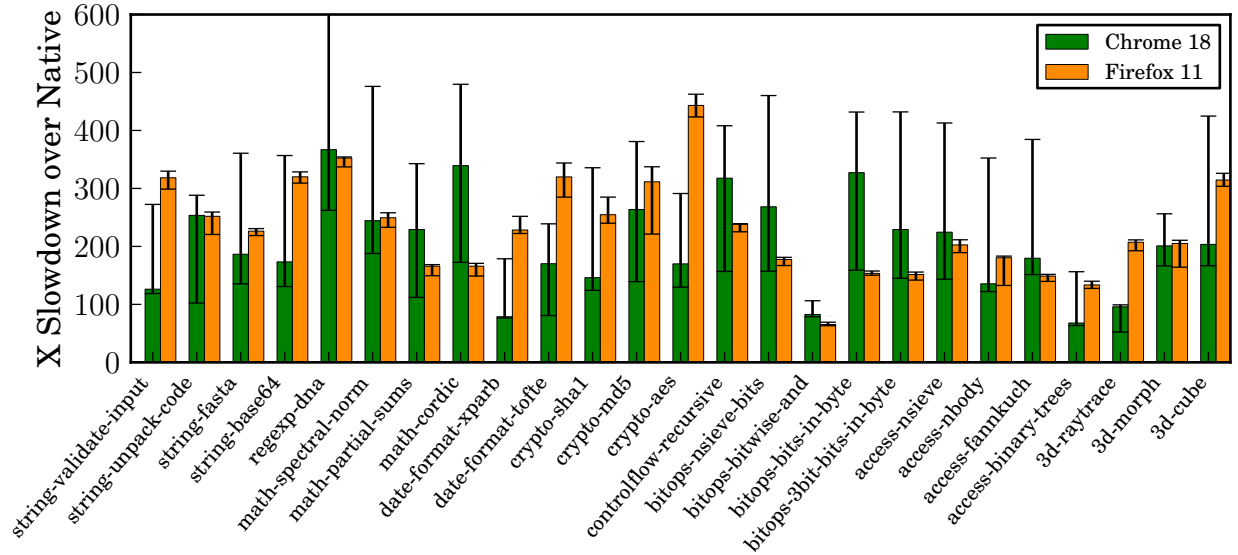


Figure 5: Relative Slowdown of js.js running the Sunspider benchmark in Google Chrome and Firefox 9. The median of ten runs of each benchmark is shown, with error bars corresponding to the minimum and maximum.

and the cross-origin policy prevents it from accessing the DOM outside of where it originates, however this approach has not fared well because today’s web pages are complex and limiting a third-party’s access to a single fixed-size iframe is not flexible enough. In addition, cross-iframe messaging is cumbersome—requiring established message-passing protocols between parties. iframes also don’t prevent page redirection, window alerts, browser denial-of-service (via spin-loop), and memory exhaustion.

There has been a lot of work [17, 15, 14, 9, 1, 7] in the area of static javascript analysis of third-party scripts to restrict content and enforce security policies. These implementations typically restrict the way the JavaScript language features are used. They enforce these restrictions by using static analysis techniques to check the parameters passed to the various functions used by the script. Since much of the policy enforcement is done statically, these solutions typically have good runtime performance. However, it is very hard to determine the security aspects of such parameters by plain parameter checking unless one does a very robust execution tracing at runtime. For example, GateKeeper [9] employs a parameter checking model, but they cannot check the safety of the complex but useful functions, such as `eval`, `setTimeout` etc., whose parameters need to be passed to the JavaScript parser. In the cases of FBJS [1] and AD-safe [17], untrusted scripts are allowed to make calls to an access-controlled DOM interface, which again supports a very restricted version of Javascript and many of these access control checks are not sound. The cost

in employing a restricted JavaScript subset is that some scripts may not conform to the subset, requiring they be ported.

Many recent techniques [16, 2, 8, 18] have taken the approach of transforming untrusted JavaScript code dynamically to interpose runtime policy enforcement checks. These works try to cover the many diverse ways in which a malicious code may subvert static policy enforcement checks. But even these policies restrict features of JavaScript (*e.g.*, `eval` and `with`) or some functions that eschew redefinition as in Browser-Shield [18]. WebSandbox [2], for example, adopts a parameter checking model to verify the parameters being passed, but additionally creates a virtualized environment for third-party scripts in which the variables have a different namespace than what is visible to the native engine. However, the arguments of functions like `eval`, when generated dynamically, would bypass such instrumentation. Since the execution is still done on the native JavaScript engine, `eval` cannot be safely executed in the WebSandbox approach.

The Google Caja project [16] enforces security policies using a mixture of static and runtime techniques. Caja provides a compiler that transforms (cajoles) the third-party script into a milder version with less capability, *i.e.*, it restricts the way a script might use the DOM API or various JavaScript constructs such as `with` and `eval`. This is done by verifying that the script adheres to the required security policy using static analysis. Where it cannot confirm that the script is well behaved, it will annotate the application with runtime checks. In con-

trast, access to the DOM with js.js is unavailable by default and has to be explicitly allowed for the third-party script to be able to access a DOM object. This negates the need for changing the script source or restricting the way the third-party applications use JavaScript APIs. Instead, js.js captures the accesses to DOM objects using callbacks and enforces a security policy at runtime. js.js also allows for pausing or terminating the execution of run-away scripts while Caja cannot handle this issue (without solving the halting problem). Caja also requires server-side execution, while js.js is client-side only.

The recent introduction of Web Workers [5] has enabled a way of sandboxing third-party scripts, but to an extreme extent. A Web Worker prevents not just a malicious third-party script but any third-party script from accessing the DOM at all. A script running in a Web Worker essentially runs in parallel to the application UI and hence can be killed at any time by the parent application that forks it, preventing loop attacks. But the forked worker has very limited functionality, having to communicate with the parent through message-passing. This requires rewriting third-party scripts, decreasing its usability.

AdJail [13], has less restriction on JavaScript functionality and adopts access control mechanisms to regulate the access to host objects. But the access control model applied in this case is not flexible enough to dictate how the object is used once a third-party script validates access to it. Our approach gives such flexibility by letting site operators build wrappers to functions that pose a security risk.

A different approach is for the website owner to ask the underlying browser to enforce the owner’s policies on any third-party JavaScript content, leaving the enforcement entirely to the browser’s discretion. Using this method, a wide variety of fine-grained security policies can be enforced with low overhead as illustrated in Content Security Policies [19], BEEP [10] and Conscript [12]. Such a collaborative approach seems sound in the long term but today’s browsers do not agree on a standard for publisher-browser collaboration, resulting in a large gap in near-term protection from malicious third-party scripts.

7 Conclusion

We have created the js.js API and runtime which allows for controlled and secure execution of untrusted JavaScript code. Our initial prototype implementation has been created by compiling the SpiderMonkey JavaScript engine to JavaScript. We then implemented the js.js API in JavaScript as a wrapper around the SpiderMonkey API. Using this API, we show secure execution of Twitter’s Tweet Button and we evaluate the per-

formance overhead of running JavaScript in JavaScript by evaluating on the SunSpider benchmark suite. In the future, we hope to both increase the performance of js.js as well as show its security potential on increasingly interesting examples.

Acknowledgements

We would like to thank Michael Freedman for his advisement and Alon Zakai for his help and excellent work on Emscripten.

References

- [1] Facebook javascript. <http://developers.facebook.com/docs/fbjs/>.
- [2] Microsoft web sandbox. <http://www.websandbox.org/>.
- [3] Spidermonkey. <https://developer.mozilla.org/en/SpiderMonkey>.
- [4] Sunspider javascript benchmark. <http://www.webkit.org/perf/sunspider/sunspider.html>.
- [5] Web workers. <http://dev.w3.org/html5/workers/>.
- [6] ANSEL, J., MARCHENKO, P., ERLINGSSON, U., TAYLOR, E., CHEN, B., SCHUFF, D. L., SEHR, D., BIFFLE, C. L., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proc. PLDI 2011*.
- [7] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for javascript. In *Proc. PLDI 2009*.
- [8] FELT, A., HOOIMEIJER, P., EVANS, D., AND WEIMER, W. Talking to strangers without taking their candy: isolating proxied content. In *Proc. SNS 2008*.
- [9] GUARNIERI, S., AND LIVSHITS, B. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proc. USENIX Security 2009*.
- [10] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proc. WWW 2007*.
- [11] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO 2004*.
- [12] LIVSHITS, B., AND MEYEROVICH, L. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proc. IEEE Security and Privacy 2010*.
- [13] LOUW, M. T., GANESH, K. T., AND VENKATAKRISHNAN, V. N. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proc. USENIX Security 2010*.
- [14] MAFFEIS, S., MITCHELL, J., AND TALY, A. Run-time enforcement of secure javascript subsets. In *Proc. W2SP 2009*.
- [15] MAFFEIS, S., AND TALY, A. Language-based isolation of untrusted javascript. In *Proc. CSF 2009*.
- [16] MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized javascript. <http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf>.
- [17] POLITZ, J. G., ELIOPOULOS, S. A., GUHA, A., AND KRISHNAMURTHI, S. ADSafety: Type-based verification of javascript sandboxing. In *Proc. USENIX Security 2011*.
- [18] REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM TWEB 2007*.
- [19] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proc. WWW 2010*.
- [20] ZAKAI, A. Emscripten: An LLVM-to-javascript compiler. <https://github.com/kripken/emscripten/blob/master/docs/paper.pdf>.