Typed-Based Verification of Web Sandboxes

Joe Gibbs Politz
Brown University
Providence, RI 02912

Arjun Guha*
University of Massachusetts
Amherst, MA 01003

Shriram Krishnamurthi
Brown University
Providence, RI 02912

February 22, 2014

Abstract

Web pages routinely incorporate JavaScript code from third-party sources. However, all code in a page runs in the same security context, regardless of provenance. When Web pages incorporate third-party JavaScript without any checks, as many do, they open themselves to attack. A third-party can trivially inject malicious JavaScript into such a page, causing all manner of harm. Several such attacks have occurred in the wild on prominent, commercial Web sites.

A *Web sandbox* mitigates the threat of malicious JavaScript. Several Web sandboxes employ closely related language-based techniques to maintain backward-compatibility with old browsers and to provide fine-grained control. Unfortunately, due to the size and complexity of the Web platform and several subtleties of JavaScript, language-based sandboxing is hard and the Web sandboxes currently deployed on major Web sites do not come with any formal guarantees. Instead, they are routinely affected by bugs that violate their intended sandboxing properties.

This article presents a type-based approach to verifying Web sandboxes, using a JavaScript type-checker to encode and verify sandboxing properties. We demonstrate our approach by applying it to the ADsafe Web sandbox. Specifically, we verify several key properties of ADsafe, falsify one intended property, and find and fix several vulnerabilities, ultimately providing a proof of ADsafe's safety.

1 Introduction

Many popular Web sites incorporate content from external, independent sources: e.g., Facebook embeds third-party games, My Yahoo! and iGoogle embed third-

*Corresponding author. Tel: +1 (413) 545-2447, Email: arjun@cs.umass.edu.

party widgets, and so on.[1] Moreover, such content is much more common than some Web users may realize. Much of the commercial Web's business model is driven by advertising, and these ads are typically served by third-party ad networks. Therefore, when a user visits a "single" Web site, such as a newspaper, that site often triggers the browser to fetch content from ad networks and other sites [64].

This external content includes not just HTML markup, images, and cookies, but third-party JavaScript too. If third-party JavaScript is naively included on a page, it runs in the same trust-domain as the page itself. This allows embedded third-party code to easily attack Web pages, by including malicious code that does all manner of harm. For example, using JavaScript, an attacker can steal cookies to hijack sessions, track which links the user follows, read password fields, completely and subtly alter the content of a Web page (e.g., planting false stock news into a financial newspaper), and more.

Many Web pages do trust third-parties and include JavaScript hosted on third-party servers. In some cases, this trust is warranted, but many third-parties—who may be trustworthy on their own—include JavaScript from other less-trustworthy parties. For example, an ad network allows anyone with a credit card to upload JavaScript-based ads. Although a Web site may trust a major advertising network, it is unwise to trust all the advertisers who use them. In fact, there have been several incidents where major sites have been compromised by malicious ads that were hosted on trustworthy advertising networks. For example, a third-party ad took over the entire New York Times Web site and encouraged visitors to buy fake anti-virus software [59]. MSNBC and several other sites were similarly compromised, when attackers uploaded a malicious ad onto Google's DoubleClick network [7]. Notably, several of these attacks start with malicious JavaScript but escalate to drive-by downloads of native binaries. Therefore, malicious JavaScript affects not just Web sites, but can be used as a vector to harm users' systems too.

There are several mechanisms that one could use to safely embed third-party JavaScript. As a first attempt, a page may try to isolate third-party code using an *IFrame*, which loads content in an isolated, embedded frame within a page. However, JavaScript within an IFrame can still open windows, communicate with servers, and perform other operations that a Web sandbox disallows. Furthermore, inter-frame communication is difficult when desired. Other mechanisms that do work include proposed changes to the IFrame mechanism, other browser modifications, browser plug-ins, and abandoning JavaScript to program in more secure languages that compile to JavaScript. We discuss these other approaches in related work (section 11). The focus of this article is on a technique known as *language-based Web sandboxing*.

Language-based Web Sandboxing There are several *Web sandboxes* for JavaScript that make it safe to embed untrusted, third-party code in Web pages.

[1] Some authors call these *mashups*. The precise terminology is open to interpretation and irrelevant to the content of this paper, so we do not use this term further.

ADsafe [12], Caja [46], FBJS [18], and Microsoft Web Sandbox [1] are some of the most prominent examples of these. Using the Google Caja sandbox (and its evolution, Secure ECMAScript or SES), Google Sites and Google Docs allow end users to run their own JavaScript code to script Web sites and Web-based spreadsheets, presentations, and documents. Hotmail Active Views, which is based on the Microsoft Web Sandbox, allows third-parties to create interactive emails using sandboxed JavaScript.

All these Web sandboxes employ broadly similar security mechanisms: they use both static and dynamic checks to ensure that untrusted JavaScript behaves "safely". Despite introducing these additional checks, they also strive to provide a familiar programming experience that is almost identical to programming ordinary JavaScript.

Unfortunately, Web sandboxing is a brittle technology and there have been a stream of sandbox-breaking bugs reported.[2] These bugs occur because Web sandboxes have to address several complex issues simultaneously. First, the correctness of most Web sandboxes depends on a delicate interplay between several static and dynamic checks; just one omitted check can break the entire sandbox. Second, Web sandboxes have to correctly attenuate access to the Web browser API that both provides rich functionality and suffers incidental complexity; it is difficult to ensure that all API calls are safe and called with safe parameters. Finally, Web sandboxes have to sandbox JavaScript code; JavaScript may have a small and simple core, but the full programming language is large, powerful (e.g., unfettered mutability), and has a number of corner-cases (e.g., rampant overloading), that make it a poor fit for strong formal reasoning.

The issues above make building and verifying Web sandboxes a challenge. The result is that Web sandboxes are developed using ad hoc techniques and have no formal guarantees.

Web Sandbox Verification This paper presents an approach to Web sandbox verification that we have successfully applied to the ADsafe Web sandbox, finding and fixing sandbox-breaking bugs along the way. We chose ADsafe for two reasons. First, it is intended as a real sandbox, and hence deals with numerous real-world subtleties, thus providing a vigorous test of our tools. Second, several previous projects had already studied it; this provided us with a benchmark to see, for instance, whether we could reproduce the errors they found (section 10.2), find new vulnerabilities (section 10.1), and to relate our work to past characterizations of vulnerabilities and properties in the same sandbox (section 11). Nevertheless, because the several Web sandboxes have similar designs [40, 42], we firmly believe our techniques are applicable to other sandboxes too (section 12).

We make the following contributions:

[2] For example, a series of results in academic papers revealed weaknesses in FBJS and ADsafe[40–43], and the Caja team has disclosed at least a dozen sandbox-violating bugs, an archive of which is at `https://code.google.com/p/google-caja/w/list?can=2&q=security+advisory`.

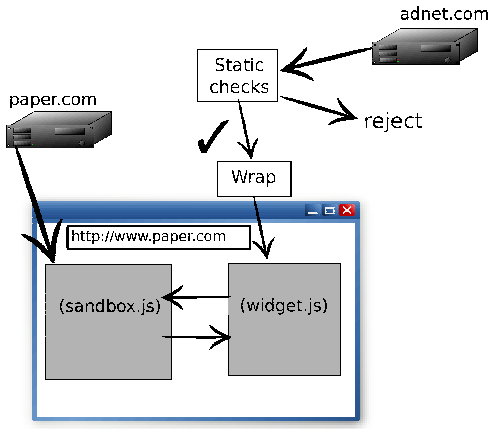

Figure 1: Web sandboxing architecture

- We provide a new characterization of ADsafe as two inter-dependent components: a type-checker and a reference monitor. This characterization is the basis of our verification methodology.
- We define what it means for ADsafe to be "safe", first informally in prose and then formally using types.
- We present a type-based approach to verifying both components of the sandbox. In particular, we start with an existing, sophisticated type system for JavaScript, but show that some simple, sound modifications make verification feasible.
- We uncover several new bugs in ADsafe and are able to detect old bugs. Notably, these bugs manifest as type errors.

Our type checker and several auxiliary files are available online at

`http://cs.brown.edu/research/plt/dl/adsafety/v2/`

2 Language-based Web Sandboxing

Web browsers provide rich APIs to JavaScript code for network access, disk storage, geolocation, and so on. In fact, browsers constantly add new APIs to give Web applications access to more user data [47]. If used legitimately, these APIs allow programmers to write rich applications. Malicious code can, however, exploit these APIs to cause harm. A Web sandbox attenuates or outright prevents access to these APIs, allowing pages to safely embed untrusted programs. These untrusted programs are commonly referred to as *widgets*.

ADsafe [12], Caja [46], FBJS [18], and BrowserShield [53] are *language-based* sandboxes that employ broadly similar security mechanisms. Maffeis, Taly, and Mitchell first characterized these mechanisms as follows [40]:

- A Web sandbox includes a static code checker that *filters* out certain widgets that are almost certainly unsafe. This checker is run before the widget code is executed by the browser.
- A Web sandbox provides runtime *wrappers* that attenuate access to the Web browser's APIs. These wrappers are defined in a trusted runtime library that is linked with the untrusted widget.
- Static checks are necessarily conservative and can reject benign programs. To avoid being overly restrictive, Web sandboxes *rewrite* possibly-unsafe programs to employ the wrappers, which perform checks dynamically.

Figure 1 illustrates this architecture. In the figure, an untrusted widget from `adnet.com` is embedded in a page from `paper.com`. The widget (`widget.js`) is filtered by the static checker. If static checking passes, the untrusted widget is rewritten to invoke the wrappers (`sandbox.js`) in the trusted runtime library. Both the runtime library and the checked, rewritten widget must be hosted on a site trusted by `paper.com`, and are assumed to be free of tampering.

Reference Monitors We observe that a Web sandbox can also be characterized as a *reference monitor* between the untrusted widget and the browser APIs. Anderson's seminal work on reference monitors identifies their certification demands [4, p 10-11]:

> The proof of [a reference monitor's] security requires a verification that the modeled reference validation mechanism is tamper resistant, is always invoked, and cannot be circumvented.

Therefore, a Web sandbox must come with a precisely stated notion of security, and a proof that its static checks and runtime library correctly maintain security. The end result should be a quantified claim of safety over *all* possible widgets that execute against the runtime library.

Our technique obeys this plan. But, as we shall see, it is not enough to simply verify the ADsafe reference monitor. This essential detail becomes clear on a close examination of the reference monitor code.

3 Code-Reviewing Web Sandboxes

Imagine we are given a Web sandbox and asked to determine (1) what guarantees it tries to provide and (2) if it does so correctly. If, like ADsafe, the system does not come with a precise, formal statement of its guarantees, we might begin with a code-review. In this section, we discuss the results of our first code-review of ADsafe. This review guides the verification we do in the rest of this article.

ADsafe, like most Web sandboxes, has two interdependent components (fig. 2 shows fig. 1 specialized for ADsafe):

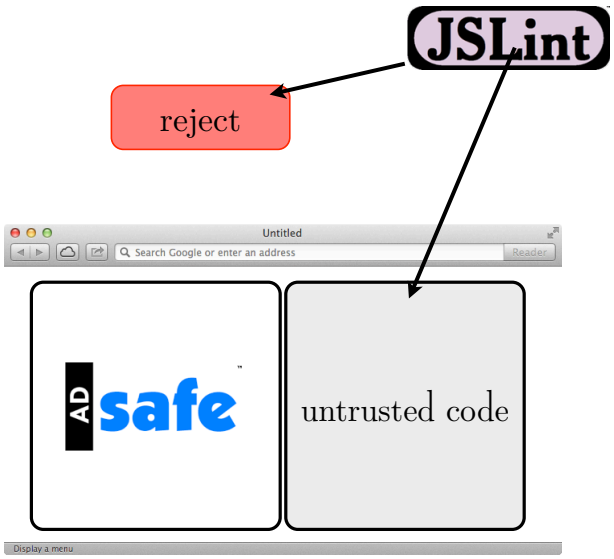

Figure 2: Architecture of ADsafe

- a static checker, called JSLint,[3] which filters out widgets not in a safe subset of JavaScript, and
- a runtime library, `adsafe.js`, which implements wrappers for the Web browser API and other runtime checks.

These two components conspire to make it safe to embed untrusted widgets, though "safe" is not precisely defined. The code review in this section will help uncover ADsafe's notion of safety, which we will state in section 4.

Attenuated Access to the Web Browser API Web browsers have several object-oriented APIs that give JavaScript programs access to a variety of services. For example, the *DOM API*[4] provides the ability to read and write anything on the Web page, the *XMLHttpRequest and WebSockets APIs* provide network access, the *Geolocation API* provides access to users' physical locations, and the *local storage and file system APIs* provide disk access. There are too many APIs to enumerate them all.

A key feature of ADsafe, which is immediately apparent on inspecting its code, is that does not give untrusted widgets direct references to any of these APIs. Direct references are dangerous because most Web browser objects have

[3]Readers familiar with JSLint may know that it performs several kinds of checks. The checks we discuss are enabled by an ADsafe-specific flag.

[4]Document Object Model

references that lead back to the root of the object graph, which is the `window` object. For example, if `elt` is a reference to any element on the page (i.e., a DOM element), a program can get a reference to `window`, which has references to all Web browser APIs:

```
var myWindow = elt.ownerDocument.defaultView;
myWindow.XMLHttpRequest;
myWindow.localStorage;
myWindow.geolocation;
```

Therefore, most of the code in ADsafe is concerned with sandboxing the DOM API. If the DOM sandbox were circumvented, the widget could get unfettered access to all browser APIs.

ADsafe provides *wrapped* references to the DOM, called *Bunches*. Bunches form the bulk of the `adsafe.js` reference monitor and they have several dynamic checks and programming patterns that need to be carefully verified:

- The reference monitor manipulates DOM references internally, but returns them to the widget wrapped in Bunches. We must verify that all returned values are in fact Bunches, and that the monitor cannot be tricked into returning a DOM reference.

- Bunches invoke DOM methods on behalf of the widget, such as `appendChild` and `removeChild`, which take DOM references as arguments. We must verify that the runtime cannot be tricked with a maliciously crafted object that mimics the DOM interface and steals references.

- The monitor attaches DOM callbacks on behalf of the widget. But, the Web browser invokes callbacks directly and passes them event objects that include DOM references. For example, a mouse-click event object would have a reference to the element that received the click. We must verify that these event objects are also correctly wrapped before they are passed to untrusted callbacks in the widget.

- ADsafe designates a sub-tree of the DOM for the widget to manipulate; the widget is not permitted to access elements outside this subtree. But, ADsafe provides wrappers for several tree-traversal methods, too. These wrappers have runtime checks to ensure that the widget doesn't get a reference (DOM or Bunch) to any element outside the subtree. We must verify that these runtime checks are correct.

- Bunches provide access to DOM functions that are only conditionally safe, such as the `createElement` function. This function can create any kind of DOM element, such lists, paragraphs, and bold-face text, that is safe. But, it can also create unsafe elements, such as `<script>` elements that load arbitrary new JavaScript. Similarly, Bunches allow widgets to set CSS styles, but a CSS URL-value can also load external code. We must verify that the arguments supplied to these DOM functions are safe.

```
function safeDiv(x, y) {
  if (y !== 0) {
    return x / y;
  }
  else {
    throw "violation";
  }
}
```

(a) The wrapper.

```
var checked = false;
var bad = {
  valueOf: function() {
    if (checked) {
      return 0;
    }
    else {
      checked = true;
      return 1;
    }
  }
};

safeDiv(10, bad);
```

(b) The attack.

```
function safeDiv(x, y) {
  if (typeof x === "number" &&
      typeof y === "number" &&
      y !== 0) {
    return x / y;
  }
  else {
    throw "violation";
  }
}
```

(c) The fixed wrapper.

Figure 3: A buggy wrapper and a widget that attacks it

ADsafe exposes twenty Bunch-manipulating functions to the widget that face all the issues enumerated. These functions depend on several private helper functions that also demand verification.

ADsafe also needs to ensure that a widget's static HTML and CSS are safe. Both HTML and CSS can contain embedded JavaScript in a number of places (for example, event attributes like `onclick`). To restrict widget behavior, JSLint ensures that all JavaScript is only found in a single script tag, and not embedded in other contexts. We do not model JavaScript embedded in these different contexts, or model HTML and CSS. It is a limitation of our verification that we only model JavaScript in the top-level context of the page. Despite this limitation, our type-checker did find a bug related to inserting unsafe values into CSS via JavaScript, which manifested itself as a type error in a runtime regular-expression test (section 10).

JavaScript Semantics There are several features and warts of JavaScript that make it even harder to build DOM wrappers such as ADsafe Bunches:

- Some JavaScript features are unsafe to use in widgets. For example, the following code obtains a reference to `window`, the root of the object graph, so it is rejected by JSLint:

  ```
  f = function() { return this; };
  var myWindow = f();
  ```

  The unsafe feature used above is the `this` variable, which is a reference to `window` within function calls. (Within method calls, `this` behaves as one would expect.) JSLint rejects widgets that uses the `this` keyword and several other unsafe features. We have to characterize the sub-language of JavaScript that JSLint admits and ensure that it does not violate the assumptions of `adsafe.js`.

ADSAFE	:	`ADSAFE.get(obj,name)`
dojox.secure	:	`get(obj,name)`
Caja	:	`$v.r($v.ro('obj'),$v.ro('name'))`
WebSandbox	:	`c(d.obj,d.name)`
FBJS	:	`a12345_obj[$FBJS.idx(name)]`

Figure 4: Similar rewritings for `obj[name]`

- Many JavaScript operators and functions include implicit type conversions and method calls that are difficult to reason about. Consider the `safeDiv(x,y)` function in fig. 3a, which is a wrapper around JavaScript's division operator. This function has two arithmetic expressions: the expression `y != 0` is trying to ensure that `x / y`, does not divide by zero. However, this wrapper suffers a common security bug.

  The attack in fig. 3b invokes `safeDiv`, passing an object instead of a number as the `y` argument. This triggers two implicit methods calls within `safeDiv`: `y.valueOf() != 0` and `x / y.valueOf()`. The attack is crafted to first return `1`, which passes the runtime check, and then return `0`, which breaks the wrapper. One way to fix the wrapper is to also assert that `typeof x === 'number'` and `typeof y === 'number'`, shown in fig. 3c.

  There are several functions in `adsafe.js` with similar checks that may suffer similar bugs. In addition, `adsafe.js` and JSLint also ensure that widgets cannot define their own `valueOf` and `toString` methods.

These and other JavaScript quirks confound even the most experience programmers. Indeed, the bugs we discover (section 10) are not deep design errors, but bugs in code to circumvent JavaScript's semantic quirks.

JavaScript Encapsulation For JavaScript objects, all fields are public. The language has no notion of private fields. If object operations are not restricted, a widget could access built-in prototypes (via the `__proto__` field) and modify the behavior of the container. Web sandboxes statically reject such expressions:

```
obj.__proto__
```

There are several other fields that are also *blacklisted* by sandboxes. However, syntactic checks alone cannot determine whether computed field names are unsafe:

```
obj["__pro" + "to__"]
```

Therefore, widgets are instead rewritten to use runtime checks that restrict access to these fields. As fig. 4 shows, different Web sandboxes rewrite field accesses in similar ways. Some sandboxes insert these and other checks automatically, giving the illusion of programming in ordinary JavaScript. ADsafe does not provide tool support for the rewriting, so in practice it is done manually. JSLint will reject programs that have not been properly rewritten to use these checks.

Web sandboxes use this technique to also simulate private fields. For example, ADsafe stores direct DOM references in the `__nodes__` field of Bunches, and blacklists the `__nodes__` field.

The Reviewability of Web Sandboxes

We have highlighted a plethora of issues that a Web sandbox must address, with examples from ADsafe. Although ADsafe's source follows JavaScript best practices, the sheer number of checks and abstractions make it difficult to review. There are approximately 50 calls to three kinds of runtime assertions, 40 type-tests, 5 regular-expression based checks, and 60 DOM method calls in the 1,800 LOC `adsafe.js` library. Various ADsafe bugs were found in the past and this article presents a few more (section 10). Note that ADsafe is a small Web sandbox relative to larger systems like Caja.

The Caja project asked an external review team to perform a code review [5]. The findings describe many low-level details that are similar to those we discussed above. In addition, two higher-level concerns stand out:

- "[Caja is] hard to review. No map states invariants and points to where they are enforced, which hurts maintainability and security."
- "Documentation of TCB is necessary for reviewability and confidence."

These remarks identify an overarching requirement for any review: the need for specifications. Specifications both tell us whether a given system will meet our needs, and provide a target against which to determine whether the implementation is correct.

4 Verification Roadmap

Defining Safety Because humans are expensive and error-prone, and because the code review needs to be repeated every time the program changes, it is best to mechanize the review process: we should use simple verification tools that do not require onerous annotations or extensive changes to ADsafe. These tools should be robust to simple changes to ADsafe as the code evolves.

However, before we begin, we need to define what security means. We present a definition that is specific to ADsafe, though the properties are similar to the goals of other Web sandboxes. From our code review (section 3) we posited some properties that we sent to the author of ADsafe, who amended and enhanced them. They are given below, rewritten slightly to use the terminology of this article:

Definition 1 (ADsafety) *If the containing page does not augment built-in prototypes*[5] *and all embedded widgets pass JSLint, then:*

[5]This requirement was highlighted by Maffeis and Taly [43].

1. *widgets cannot load new code at runtime, or cause ADsafe to load new code on their behalf;*

2. *widgets cannot affect the DOM outside of their designated subtree;*

3. *widgets cannot obtain direct references to DOM nodes; and*

4. *multiple widgets on the same page cannot communicate.*

We emphasize that, while there are many reasonable variations on this set of properties, these were the ones defined by the author of ADsafe, and were hence taken by us to be definitional for the system.

Note that the first two properties are common to sandboxes in general—allowing arbitrary JavaScript to load at runtime compromises all sandboxes' security goals, and all sandboxes provide mediated access to the DOM. We also note that the assumption about built-in prototypes is often violated in practice [19]; many Web pages do extend builtin prototypes in a manner that breaks ADsafe's assumptions. ADsafe is not designed to check for such problems, so we cannot verify that it does. To address this issue, one would have to add new features to ADsafe, as Finifter et al. [19] do, but our goal is to only fix bugs that violate ADsafety as defined by the author of ADsafe.

The ADsafety definition is precise, but it is informal. A mechanized verification requires a formal definition of ADsafety, which depends on a formalism for JavaScript and our choice of verification technology.

Verification Technology: Type Checking There are many ways to verify JavaScript code, including program analysis [14, 22, 25, 31], program logic [21], and type checking [10, 27, 51, 57]. We employ a general purpose JavaScript type-checker [37] to define and verify ADsafety. Type-checking is well-suited for this task for several reasons. First, programmers are familiar with type systems, and ours is mostly standard (the novelties are discussed in section 6). This lessens the burden on sandbox developers who need to understand what the verification is saying about their code. Second, our type system is reasonably efficient, leading to a quick procedure for checking ADsafe's runtime library (20 seconds). Efficiency and understandability allow for incremental use in a tight development loop. Finally, our type system is accompanied by a soundness proof. This property enables the actual verification. Thus, the features of comprehensibility, efficiency, and soundness combine to make type checking an effective tool for verifying some of the properties of Web sandboxes.

In section 9, we make type-based arguments to prove an ADsafety theorem. This is only possible after fixing the bugs our type-checker discovers, which we detail in section 10.

Verification Tasks Since our goal is to verify ADsafe using types, we might try to just type-check the reference-monitor code (`adsafe.js`). However, on its own, this will surely fail because the `adsafe.js` reference monitor *does not monitor arbitrary JavaScript code.* It is only ever linked to code that has passed

the static checks of JSLint. In fact, if the JSLint checks are skipped, it becomes trivial to circumvent the reference monitor. Therefore, we have to first understand what it means for code to pass JSLint's static checks.

JSLint is a lint-like tool that applies ad hoc syntactic restrictions. For verification, we have to characterize the semantic consequences of these restrictions. For example, JSLint signals errors when code is poorly-indented (as linting tools do). Indentation has no semantic consequence, but JSLint has other restrictions that matter for security. We must understand the nature of these restrictions and ensure they are correctly implemented by JSLint. In our work, we use static types to capture these restrictions.

Therefore, we have two distinct tasks: to characterize JSLint (section 7) and to then use this characterization to verify ADsafe's reference monitor (section 8). We will use types as the basis for both tasks (section 6). However, we must first address an even more fundamental problem: we need a semantic model of JavaScript that is suitable for building a type checker and proving type soundness.

5 Modeling JavaScript

A type-based verification of ADsafe, or any Web sandbox, requires type-checking JavaScript code. Unfortunately, JavaScript is a large language and has several tricky corner cases in its semantics. Combined, these make it very difficult to build a type-system for JavaScript (let alone prove it sound). The JavaScript specification [15] is not helpful, since it simply codifies the complexity of the language: it is a 200+ page document of prose and pseudocode (along with several ambiguities).

Maffeis, Mitchell, and Taly [39] have distilled the JavaScript specification into a much smaller (30-page) operational semantics. Their semantics closely follows the specification and, by doing so, finds several ambiguities and inconsistencies in the specification itself. For our purposes, however, this semantics is problematic because we require a semantics that is small enough to be amenable to type-checking and proofs of soundness. Each additional syntactic form and each new semantic rule makes building the type-checker and doing proofs harder. Therefore, we need a semantics that can "shrink" the language.

We use the λ_{JS} semantics of of Guha et al. [26], which was designed to achieve these goals. λ_{JS} is a small language that models the essentials features of JavaScript:[6] prototype-based objects, first-class functions, mutable state, and a few simple control operators. The full syntax of λ_{JS} is presented in fig. 5. It is significantly smaller than the syntax of JavaScript itself, which runs for several pages. Since there are so few syntactic forms, there are commensurately few semantic rules (fewer than 30 rules). It is thus much easier to build tools and

[6] λ_{JS} models ECMAScript 3 [15], which is the target of our verification. Modern Web browsers conform to ECMAScript 5 [16], which adds several new features and breaks compatibility with ECMAScript 3 in some corner cases.

v	=	*num*	numbers
	|	*str*	strings
	|	*bool*	boolean values
	|	**undefined**	a special JavaScript constant
	|	**null**	a special JavaScript constant
	|	**func**$(x_1 \ldots x_n).e$	functions of arity n
	|	`{` str`:`$v\ldots$ `}`	objects
e	=	x	variables
	|	v	values
	|	**let** `(`x `=` e_1`)`e_2	let bindings
	|	e_f`(`$e_1 \ldots e_n$`)`	function application
	|	e`[`e`]`	lookup object member, or **undefined** if not found
	|	e_1`[`e_2`]` `=` e_2	update object member, or create if not found
	|	**delete** e`[`e`]`	remove object member; no error on failure
	|	e `=` e	heap cell update
	|	**ref** e	heap cell allocation
	|	**deref** e	heap cell dereference
	|	**if** `(`e`)` e **else** e	conditional
	|	e`;`e	sequence
	|	**while**`(`e`)` e	loop
	|	$label$`:` e	labeled expression
	|	**break** $label$ e	break to label
	|	**try** e **catch** `(`x`)` e	catch exception
	|	**try** e **finally** e	finally block
	|	**throw** e	throw exception
	|	op_n`(`$e_1 \ldots e_n$`)`	primitive operator

Figure 5: Syntax of λ_{JS} (adapted from Guha et al. [26])

do proofs for λ_{JS}. Guha et al. also define a *desugaring* function[7] that translates JavaScript code to λ_{JS}.

The λ_{JS} semantics was designed based on an intimate knowledge of the language, and intends to accurately mirror the specification. Nevertheless, errors can occur given the size of the specification, the number of corner-cases in the language, and the fact that we are mapping JavaScript to a smaller language. Unfortunately, the specification is an English document, so it is not clear how one can automatically check *any* semantics for conformance to it.

It is, however, easy to define an evaluator for λ_{JS}, and the composition of the evaluator with desugaring results in another evaluator for source JavaScript terms. This can then be compared against real browser implementations for

[7]Technically this is a syntax-directed compiler, because it is translating JavaScript to a slightly different language. However, the term "desugaring" is evocative of the intended purpose and nature of this compiler.

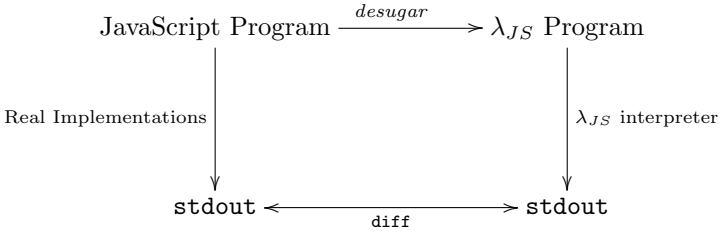

Figure 6: Testing strategy for λ_{JS} [26]

consistency (fig. 6). The λ_{JS} paper substantiates the following claims for a portion of the Mozilla test suite:

Claim 1 (Desugaring is Total) *For all JavaScript programs e, desugar*$[\![e]\!]$ *is defined.*

Claim 2 (Desugar Commutes with Eval) *For all JavaScript programs e,* $desugar[\![eval_{JavaScript}(e)]\!] = eval_{\lambda_{JS}}(desugar[\![e]\!])$.

In principle, of course, this process should be applied to every JavaScript evaluator; doing so is mainly a matter of labor. In practice there are situations where individual evaluators either behave differently from one another, or even do not conform to their own test suites. Deciding what to do in these cases is a function of the intended purpose. In security settings, matching the behavior of browsers over the standard may be wise, as attackers attack implementations, not specifications. Recognizing the problems posed by browser variations, however, the ECMAScript community now publishes its own conformance suite, and a successor of λ_{JS} [49] now matches that, thereby further erasing the gap between the specification and the tested semantics.

This testing strategy, and the simplicity of λ_{JS}, helps us gain confidence in the tools discussed in this article. In fact, for the same reasons, other authors have also employed λ_{JS} to build JavaScript verification tools [10, 20, 57].

6 Type-Checking JavaScript

Our verification approach is to use a sophisticated type-system for JavaScript to express and verify security properties. In earlier work, we proved our type-checking techniques sound [23, 27, 51]. However, to verify ADsafe, we have to make one additional modification to the previously documented type system: we relax several typing rules to allow more runtime errors [52]. In this section, we illustrate why this modification is necessary and argue that it is safe. We begin with an overview of our type system's other key features.

S, T	$=$	$\mathsf{Num} \mid \mathsf{Str} \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Undef}$	base types
	$\mid$	$T_1 \cup T_2$	union types
	$\mid$	$T_1 \cap T_2$	intersection types
	$\mid$	$\mu\alpha.T$	recursive types
	$\mid$	$\mathsf{Ref}\ T$	heap reference type
	$\mid$	$S_1 \times \ldots \times S_n \to T$	function type
	$\mid$	$S_1 \times \ldots \times S_n \times S_{\text{rest}} \cdots \to T$	variable-arity function type
	$\mid$	$\{L_1^{p_1} : T_1 \ldots L_n^{p_n} : T_n, L_A : \mathbf{abs}\}$	object types
L	$=$	$\ldots$	regular expressions
p	$=$	$\downarrow$	field definitely present
	$\mid$	$\circ$	field possibly absent

Figure 7: Types for λ_{JS}

6.1 Type System Features

Our JavaScript type-checker works by first desugaring JavaScript to λ_{JS}, then type-checking the resulting λ_{JS} program. As discussed in section 5, this design makes it easier to engineer the type-checker and to be assured that it is correct. Our system requires some type annotations,[8] which we insert into the source program as JavaScript comments. Concretely, any comment that begins with a colon is parsed as a type annotation:

```
/*: Num */
/*: Bool -> Bool */
```

This design has a significant engineering benefit: we can simply run the typed program unmodified in the browser. Formally, this means that our type system satisfies a simple *type erasure* property.

Our type language has several features [37], but the fragment of our types necessary for verification is presented in fig. 7. Most of the types are conventional, but it includes two unusual features: (i) untagged union types, which we discriminate with *flow typing* [27] and (ii) support for objects with *first-class member names* [51]. Both these features are necessary to type-check programming patterns that ADsafe employs liberally.

Flow Typing JavaScript has powerful reflection mechanisms that programs, including `adsafe.js`, use to inspect the structure of values at runtime. In contrast to languages like ML and Haskell, which support pattern matching, JavaScript programs use arbitrary control operators and mutable state to discriminate values.

Figure 8 shows a few patterns that JavaScript programs use:

a. An *if-split* [55] is a simple, syntactic type-test: the if-condition tests the type of a variable `x`, and the type of `x` thus differs in each branch.

[8] We use *bidirectional typechecking*, which is able to infer many annotations.

```
/*: Num ∪ Str → Undef */
function(x) {
  if (typeof x === "number") {
    // x is a number
    ...
  }
  else {
    // x is a string
   ...
  }
}
```

(a) A simple, syntactic *if-split*

```
/*: Num ∪ Str → Undef */
function(x) {
  if (typeof x === "number") {
    // x is a number
    ...
    return;
  }
  // x is a string
  ...
}
```

(b) Using `return` and fall-through

```
/*: Num ∪ Undef → Undef */
function(x) {
  if (typeof x === "undefined") {
    // x is the value undefined
    x = 0;
  }
  // x is a number: the body of the
  // if-statement sets x to 0; if
  // the body is skipped, the type
  // annotation states that x is
  // a number.
  ...
}
```

(c) Using state and control

```
/*: Str ∪ Bool ∪ Num → Undef */
function(x) {
  switch (typeof x) {
    case "boolean":
    case "number":
      // x is either a boolean or a
      // number, due to fallthrough
      ...
      break;
    default:
      // x is a string, since other
      // cases are handled above.
      ...
  }
}
```

(d) Using `switch` and `case`-fall-through

Figure 8: A few patterns for discriminating types in JavaScript

b. JavaScript has several statements that alter the normal flow of control. Consider testing if `x` is a number, and then using **return** to abort the function if the test holds. Then, when the test does not hold, the function does not abort, and we can narrow the type of `x` in the rest of the function.

c. A program can also use state to change the type of a variable. A common JavaScript pattern is to test if a variable `x` is the special value **undefined**[9] and then set it to a default value with a different type.

d. Finally, JavaScript has several other control operators, including the **switch** statement, which supports C-like fallthrough when cases elide **break**. Programs rely on these features to discriminate types, too.

To avoid heavy modifications to ADsafe, we have to be able to type-check code that uses control and state to reason about types. We do so using flow typing [27], which augments type checking with a control-flow analysis. The control-flow analysis runs *locally* on each function (i.e., it is intraprocedural). It is not as powerful as whole-program static analyses [22, 25, 31]. However, it is powerful enough to tackle ADsafe and runs with reasonable speed.

Objects with First-Class Member Names JavaScript uses prototype inheritance, and thus has no notion of classes. Instead, all objects can be freely modified: fields can be arbitrarily created, updated, and even deleted. Such modifications aren't just possible, they are also easy, because JavaScript has a convenient dictionary notation for accessing and updating objects. In earlier work, we developed a type system that is able to type-check JavaScript objects [51] which we summarize below.

The key to typing objects in ADsafe is to use regular expressions to ascribe types to collections of fields. For example, the following type says that fields beginning with `"x"` are numbers:

$$\{\texttt{x.*} : \mathsf{Num}\}$$

However, in the type above, it is not clear whether an object contains all such fields (i.e., has an infinite set of fields) or just some fields of that type. To resolve this ambiguity, our type system uses *presence annotations* on field names:

- $x^{\downarrow} : T$ indicates that the field x is *definitely present* and has type T.
- $x^{\circ} : T$ indicates that the field x is *possibly present* (thus possibly absent too). If it is present, it has type T. Note that x may be a regular expression that denotes a set of possibly present fields.
- **abs** : x indicates that the field x is *definitely absent*. Again, x may be a regular expression that denotes a set of absent fields.

[9] When a function's argument is elided, JavaScript does not signal arity-mismatch errors, but instead sets missing arguments to the value **undefined**, thereby supporting a lightweight form of variable arity.

For example, in the following type, `y` is definitely present, the collection `x.*` is possibly present, and `z` is definitely absent:

$$\{\mathsf{y}^{\downarrow} : \mathsf{Bool}, \mathtt{x.*}^{\circ} : \mathsf{Num}, \mathsf{z} : \mathbf{abs}\}$$

Despite these extensions, our object types are conventional in the following key way: any field not mentioned in a type is inaccessible. Thus, types can be used to hide fields by subsumption. Information hiding using types is key to our approach.

6.2 Type Errors and Type Soundness

In earlier work, we established the soundness of our type-checking techniques [23, 27, 51]. However, to check ADsafe, we made an unusual change: we modified our type checker to allow many runtime errors to occur, instead of rejecting them statically. For example, consider the following function application expression, which has a number in function position:

```
100(9);
```

If this is evaluated, JavaScript signals an exception:

```
TypeError '100' is not a function
```

Most type-checkers, including ours, would statically reject this program, so that the dynamic error is caught statically. However, for ADsafe verification, we had to relax the type system to allow this error.

We relax our type system in this manner for two reasons. Our first reason is pragmatic: there are several expressions (such as functional applications) within the ADsafe reference monitor that can signal runtime errors. These errors are perfectly safe: they do not violate ADsafety (definition 1). In principle, we could modify ADsafe to insert additional runtime checks around these expressions. In practice, this would involve extensive modifications that would fundamentally alter ADsafe, whereas we want to analyze the system as-is.

Our second reason is fundamental: without this change, we would not be able to account for JSLint using types (section 7). The static checks in JSLint allow JavaScript runtime errors to occur within widgets. (After all, a widget can't do harm if it stops running.) If our type checker eliminated JavaScript's runtime errors too, we wouldn't be modeling JSLint at all, but a subset of JavaScript that is free of runtime errors and completely artificial.

Having made this change, however, we need to argue that it does not violate soundness. This particular relaxation—allowing numbers in function position—is perfectly sound. It would be unsound to allow functions to be applied to arguments of the wrong type; this would break type-preservation. In contrast, our modification is sound, since it is preserves types and just adds an additional, *well-defined error*. The generalization of this idea, which we call *progressive typing* (because it parameterizes the type system over the meaning of progress [62]) has also been formalized and proven sound separately [52].

```
function fac(n) {
var r = 1; // error: bad indentation
    for (var i = 1; i <= n; i++) { // error: declare variables on top
        r = r*i; // error: missing spaces around *
    }
    return r;
}
```

Figure 9: Many of JSLint's errors are irrelevant to ADsafety

Finally, instead of following our approach, which brings verification technology to an existing Web sandbox, we could have modified ADsafe extensively or just built a Web sandbox anew. Both alternatives are likely to be more amenable to verification. Yet, we believe the harder approach that we follow will be more fruitful, as it pushes the limits of existing techniques and helps distill informal, but pervasive, "best practices".

7 JSLint: Modeling A Secure Sublanguage

The ADsafe reference monitor expects to execute against widgets that have been statically checked and rewritten to pass JSLint, as shown in fig. 1. By linting widgets in this manner, the reference monitor can assume that they are written in an *implicit sub-language* of JavaScript. For verification, it is necessary to define this sub-language explicitly.

As the name suggests, many of JSLint's checks and errors are lint-like code-quality checks (fig. 9). They make programs easier for humans to understand, but are actually inconsequential to ADsafety. JSLint also checks the static HTML of a widget. For example, it ensures that the HTML doesn't load other, non-JavaScript code such as Java applets. As we mentioned earlier, verifying HTML is beyond the scope of our work; we do not discuss it further. We instead focus on the security-related static JavaScript checks in JSLint.

7.1 A Type for Widgets

The `adsafe.js` reference monitor makes several assumptions about the shape of untrusted widget values. These assumptions are not documented precisely, but they correspond to various static checks in JSLint. In this section, we give examples of several widget fragments that both pass and fail JSLint. These examples illustrate the attacks JSLint tries to thwart, the invariants it tries to maintain, and the shape of values it allows.

These examples will also help us write down a type (the "Untrusted" type) that precisely describes the shape of untrusted values. The Untrusted type, which we derive from JSLint, will then play a key role in the verification of `adsafe.js` in the next section. The full type is shown in fig. 10 and we derive it by example over the course of this section.

Primitives As a warmup, we note that JSLint admits JavaScript's primitive values: numbers, strings, booleans, and the special values **null** and **undefined**. To express the union of all these types of values, we define the union type Prim:

$$\mathsf{Prim} \equiv \mathsf{Num} \cup \mathsf{Str} \cup \mathsf{True} \cup \mathsf{False} \cup \mathsf{Null} \cup \mathsf{Undef}$$

Programs can discriminate this union using reflection, as illustrated in section 6.

Therefore, the Untrusted type subsumes Prim, but JSLint allows several other values, such as objects and functions. However, untrusted widgets cannot use objects and functions indiscriminately. JSLint restricts the use of objects and functions in several ways to maintain ADsafety.

Objects and Blacklisted Fields JSLint allows widgets to create objects, such as these:

```
var x = { "x": 34, "y": 70 };
var y = { "name": "foo" };
```

However, certain field names make JSLint signal errors. For example, JSLint rejects the following code because it uses the field name valueOf:

```
var z = { valueOf: 10 }
```

The code above is innocuous. But, recall from section 3, that a malicious widget can subvert runtime checks by setting valueOf to a stateful method (fig. 3b).

JSLint maintains a *blacklist* of fields that untrusted values cannot contain. The blacklist is as follows:

- The toString and valueOf fields are blacklisted. As we saw in fig. 3b, the valueOf method is implicitly invoked in several JavaScript contexts and can be used to circumvent `adsafe.js`. Therefore, JSLint simply prohibits widgets from defining valueOf methods. JSLint blacklists toString for the same reason: this method is also invoked implicitly in several JavaScript contexts. This makes building (and verifying) the reference monitor much easier.

- The fields arguments, caller, and callee, watch, and unwatch are blacklisted. Popular browsers define these fields for certain builtin objects. An attacker can use them to walk the JavaScript stack and even inspect stack frames for code in the reference monitor. ADsafe conservatively blacklists these fields from *all* objects.

- The field prototype is blacklisted. A malicious widget may try to modify the prototypes of builtin JavaScript objects (e.g., strings and numbers), which would change the semantics of code in the reference monitor.

- The fields eval and constructor are blacklisted. On some objects, these fields refer to the canonical eval function that can be used by an attacker to load arbitrary code (constructor can be used to get access to eval, too). The use of eval is thus problematic because it circumvents JSLint.

- The field __proto__ is blacklisted.[10] This field holds a reference to an object's prototype. If a widget could access or modify builtin prototypes, it could change the semantics of operations used within `adsafe.js`.
- All fields that begin and end with two underscores are blacklisted. More succinctly, fields that match the regular expression __.*__ are blacklisted. ADsafe uses these fields internally as "private" fields, in which it stores references to underlying DOM objects that untrusted code should not be able to directly manipulate. This hack is necessary because JavaScript itself has no notion of private fields.

We can formally specify this blacklist as a type. Using regular expressions and presence annotations, as described in section 6, we write a type that describes all fields that are *not blacklisted* as follows:

- We define a regular expression that accounts for all blacklisted fields, with the exception of toString and valueOf. These two fields require slightly different treatment, which we address below:

$$blacklist \equiv (\text{"caller"} \mid \text{"callee"} \mid \text{__.*__} \mid \ldots)$$

  Above, we abbreviate the blacklist for readability, but the full list is shown in the final type in fig. 10.

- All fields that are not blacklisted are possibly present in untrusted code:

$$\overline{blacklist}^{\circ}$$

  Above, $\overline{blacklist}$ is an extended regular expression that denotes the complement of *blacklist* and the "$\circ$" superscript is a presence annotation that indicates that non-blacklisted fields may or may not be present.

- Non-blacklisted fields may hold any untrusted value, i.e., Untrusted-typed values:

$$\left\{\overline{blacklist}^{\circ} : \mathsf{Untrusted}\right\}$$

- The valueOf and toString methods must be absent, but they cannot be blacklisted entirely:

$$\left\{\overline{blacklist}^{\circ} : \mathsf{Untrusted}, (\text{"toString"} \mid \text{"valueOf"}) : \mathbf{abs}\right\}$$

[10] The presence of "__proto__" as an accessible field (rather than a hidden internal property of objects) is actually non-standard behavior per the ECMAScript specification, but the "__proto__" field is a de facto standard in browsers. Therefore, to be pragmatic, ADsafe accounts for __proto__ and our tools model its semantics. Several years after we made this decision, it is now in the draft proposal for the next version of ECMAScript (ES6), because of its ubiquity in browsers (see July 15, 2013 version, section B.2.2 at `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`).

This type design allows `adsafe.js` to invoke the built-in valueOf and toString methods, which are safe, but prevents untrusted code from overriding them with unsafe methods. Even if `adsafe.js` does not invoke these methods explicitly, there are situations where the JavaScript runtime may invoke them implicitly (e.g., fig. 3c). This type design ensures that implicit invocations are possible and safe.

- Finally, we combine the object type above with the type of primitive values using a union type:

$$\mathsf{Untrusted} = \mathsf{Prim} \cup \mathsf{Ref} \left\{ \begin{array}{l} \overline{(\text{"caller"} \mid \text{"callee"} \mid \_\_.*\_\_ \mid \ldots)}^{\circ} : \mathsf{Untrusted}, \\ (\text{"toString"} \mid \text{"valueOf"}) : \mathbf{abs} \end{array} \right\}$$

This type shows a detail of our type system: Ref indicates that the object is mutable. Here we also explicitly introduce the recursive nature of Untrusted, which refers back to itself in the type of non-blacklisted fields.[11]

Functions JSLint permits widgets to define functions, such as the increment function below:

```
function incr(x) { return x + 1; }
```

But, a widget can then apply incr to any argument. For example, JSLint permits all the following function calls, even though some pass the wrong number of arguments and others pass non-numeric values to incr:

```
incr(5); // ok
incr(); // argument missing
incr(4,5); // extra argument
incr("{ x: "hello" }") // wrong type of argument
```

We need to give incr a type that accounts for this variety of arguments. This turns out to be straightforward to do: since the arguments are defined in widget code, they all have the type Untrusted. Therefore, the type of incr is a function that receives Untrusted-typed values and produces an Untrusted-typed value. In fact, all functions in a widget can be given exactly the same type! This observation is key, because it means that our approach, although it uses types internally, requires no type annotations in widgets themselves.

The formal type for functions exposes some of JavaScript's quirks, which we discuss next. But, these are not exposed to widget programmers. Functions in JavaScript are objects with an internal *code* field. λ_{JS} makes this explicit: it desugars functions to objects with a *code* field and desugars function application to extract the function from the *code* field. In addition, λ_{JS} makes JavaScript's this keyword an explicit first argument for all functions.

[11] We use the same name (Untrusted) on the left and right-hand sides of the type, which indicates a recursive type. The underlying theory and implementation use textbook equi-recursive types.

Therefore, in our type, we simply add the type of *code* to the object type:

$$\ldots \mathsf{Ref}\left\{\begin{array}{l} \mathit{code} : \mathsf{Global} \cup \mathsf{Untrusted} \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}, \\ \ldots \end{array}\right\}$$

The type of the *code* field indicates that widget-functions may have an arbitrary number of Untrusted-typed arguments and return Untrusted-typed results.[12] It also specifies that the type of the implicit this-argument (which is the explicit first argument in λ_{JS}) may be either Untrusted or Global—a JavaScript peculiarity we explain below.

The type of this reflects a key semantic subtlety of JavaScript: all functions have a this argument. if an application does not syntactically resemble a method call (i.e., `o.m()` vs. `f()`) then this is bound to the "global object". In browsers, the global object is window, which holds methods to access the network, query the geolocation API, etc. Therefore, it is very important that there be a type-distinction between Global and Untrusted. In particular, the type Global is not a subtype of Untrusted, which expresses the underlying reason for JSLint's rejection of all widgets that contain this.

Prototypes There are two related fields in JavaScript that interact with prototype inheritance. The first, "__proto__", holds a value that the runtime delegates to (recursively) for field lookup when fields aren't found on the object. For example:

```
var o = {__proto__: {x: 5}};
o.x; // evaluates to 5
var o2 = {__proto__: {__proto__: {y: 5}}};
o2.y; // evaluates to 5
```

JSLint does not allow widgets to explicitly manipulate objects' prototypes. However, since field lookup in JavaScript implicitly accesses the prototypes, we specify the type of prototypes in Untrusted:

$$\ldots \mathsf{Ref}\left\{\begin{array}{l} \text{"\_\_proto\_\_"} : \mathsf{Object} \cup \mathsf{Function} \cup \ldots, \\ \ldots \end{array}\right\}$$

The "__proto__" field enumerates several safe prototypes, such as Object and Function above, which denote the prototypes of all JavaScript objects and functions respectively. Notably, the full list of safe prototypes does not include any prototypes of the Web browser APIs. This ensures that a widget cannot get a direct reference to any Web browser API object, which ensures that they are sandboxed as intended (section 3).

There is a second field, "prototype", that is distinct in behavior from "__proto__", though related. Each created function, as well as every built-in function, has a "prototype" field. It usually holds functions that are intended to be shared across instances of a datatype; it also has a field "constructor", which holds a reference to the original function:

[12] The $T \cdots$ syntax is a literal part of the type, and means the function can be applied to any number of additional T-typed arguments. This is *uniform variable-arity polymorphism* [56].

```
function f() {}
f.prototype; // is defined
f.prototype.constructor === f; // is true
"a-string".charAt === String.prototype.charAt; // is true
```

JSLint blocks direct use of "prototype", because it allows code to modify the behavior of built-ins, which could change the behavior of ADsafe itself (which relies on values like String.prototype.charAt). However, JavaScript itself makes use of "prototype" when using the special **new** operator:

```
function f() {}
f.prototype; // is defined
var o = new f();
o.__proto__ === f.prototype; // is true
```

Since **new** is syntactic sugar (that λ_{JS} makes explicit in its desugaring step) for a pattern that uses the "prototype" field, we cannot blacklist "prototype" and still type-check **new** expressions; the type-checker will rightly complain that the **new** expression is accessing a blacklisted field. Instead, we give a precise type to the "prototype" field that is compatible with its behavior in **new** expressions:

$$\mathsf{NewProto} = \{\text{"}\_\_\mathsf{proto}\_\_\text{"} : \mathsf{Object}, \overline{\{\text{"}\mathsf{constructor}\text{"}\}} : \mathbf{abs}\}$$

This type directly blacklists the "constructor" field on the prototype, in keeping with its blacklisting on Untrusted in general. Since all the other fields are absent but "__proto__", and Object is already included in the set of allowed prototypes for Untrusted, this doesn't extend the set of values that can be found on a field lookup. We include NewProto in the allowed types of the "__proto__" field of Untrusted to capture objects that are created via **new** (fig. 10).[13]

Typing Private Fields Widgets cannot create fields that begin and end with underscores. However, ADsafe uses some of these fields itself as private fields to build the Bunch abstraction:

$$\ldots\,\mathsf{Ref}\left\{\begin{array}{l}\text{"}\_\_\mathsf{nodes}\_\_\text{"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef},\\ \text{"}\_\_\mathsf{star}\_\_\text{"} : \mathsf{Bool} \cup \mathsf{Undef},\\ \ldots\end{array}\right\}$$

Notably the type of "__nodes__" $\mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}$ is not a subtype of Untrusted, so widgets themselves cannot access it.

JSLint as a Type Checker The full Untrusted type in fig. 10 is a formal specification of the shape of values that widgets manipulate. The ADsafe reference monitor assumes that all the values it receives have this type and ensures that all the values it returns to widgets have this type, too.

[13] Although JSLint allows **new** expressions, their use cases are fundamentally limited by other restrictions. Without the ability to use this or prototype, **new** is no more than a function call with the side-effect of instantiating a new object. ADsafe's author cited compatibility [personal communication] with third-party libraries (not necessarily checked by JSLint) that provide interfaces to their APIs via **new** as a reason to allow it. Though this is a somewhat narrow use case, we model **new** to faithfully capture JSLint's behavior.

$$\mathsf{NewProto} = \{\text{"\_\_proto\_\_"} : \mathsf{Object}, \overline{\{\text{"constructor"}\}} : \mathbf{abs}\}$$

$$\mathsf{Untrusted} = \mathsf{Prim} \cup \mathsf{Ref} \left\{ \begin{array}{l} \overline{\left\{ \begin{array}{l} \text{"\_\_"}\overline{\{\text{"proto"},\text{"nodes"},\text{"star"}\}}\text{"\_\_"}, \\ \text{"arguments"},\text{"caller"},\text{"callee"},\text{"eval"}, \\ \text{"watch"},\text{"unwatch"},\text{"constructor"} \end{array} \right\}}^{\circ} : \mathsf{Untrusted}, \\ \mathit{code} : \mathsf{Global} \cup \mathsf{Untrusted} \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}, \\ \text{"\_\_proto\_\_"} : \begin{array}{l} \mathsf{Object} \cup \mathsf{Function} \cup \mathsf{Bunch} \cup \mathsf{Array}, \\ \cup\ \mathsf{RegExp} \cup \mathsf{Str} \cup \mathsf{Num} \cup \mathsf{Bool} \\ \cup\ \mathsf{Undef} \cup \mathsf{NewProto} \end{array} \\ \text{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \text{"\_\_star\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \text{"prototype"} : \mathsf{NewProto}, \\ \{\text{"toString"},\text{"valueOf"}\} : \mathbf{abs} \end{array} \right\}$$

Figure 10: The Untrusted type

JSLint satisfies the reference monitor's assumptions by statically ensuring that *all sub-expressions in the widget have type Untrusted*. This property is very easy to express with types and enforce with a type-checker. This is all that we need to verify of the reference monitor. But, before we do so, we should pause to ensure that our type-based characterization of JSLint is accurate.

7.2 Untrusted and JSLint Correspondence

As a matter of principle, we would like know if our characterization of JSLint as a type-checker matches the actual behavior of JSLint. That is, we would like to substantiate the following claim:

Claim 3 (Linted Widgets Are Typable) *If JSLint (with ADsafe checks) accepts a widget e, then e and all of its variables and sub-expressions can be Untrusted-typed.*

In principle, we could try to exploit the fact that JSLint is written in JavaScript and attempt to analyze its source. However, this is not a general technique because not every sandbox's static checker has this property; indeed, the checker (and rewriter) of Caja is implemented in Java. We would therefore prefer a technique that is independent of the implementation language (though, of course, it can be complemented, or even replaced, by one that analyzes source).

Therefore, we instead validate the claim of JSLint-Untrusted correspondence by testing.[14] We use ADsafe's sample widgets as positive tests—widgets that should be typable and lintable—and our own suite of negative test cases (widgets that are untypeable or unlintable). The direction of the implication is intentional: an unlintable widget may still be typable, and indeed our type checker admits widgets that JSLint rejects (as we discuss below). Thus, our testing,

[14] Testing revealed a security vulnerability in JSLint (section 10).

combined with the direction of the implication above, gives users confidence that using JSLint is an acceptable means of obtaining our security guarantee.

However, as our verification only depends on the Untrusted type, a user could use our type checker—checking for the Untrusted type—as a substitute for JSLint, and thus be able to safely use a larger set of widgets with `adsafe.js`. Doing so would also eliminate having to trust the testing process for JSLint-Untrusted conformance. Users are thus free to choose between accepting the testing (and continuing to use JSLint) or using our type-checker instead (and thus changing their workflow, but being able to write more widgets).

7.3 Differences Between JSLint and Typed Widgets

We argue that if a widget passes JSLint, then it is also Untrusted-typable. However, Untrusted-typability does not imply that the widget passes JSLint. JSLint is a "code quality tool" that was retrofitted to perform security checks. Some of these code quality checks are irrelevant for safety, so they reject safe code that the Untrusted-type admits. We give some examples below.

Banned Names The Untrusted-type states that prototype, arguments, and several other strings cannot be used as field names. However, it is safe to use them as identifiers:

```
ADSAFE.lib("Widget_", function() {
  var prototype, arguments;
});
```

JSLint rejects this program because it simply scans for "banned names".

Unused Identifiers JSLint requires identifiers to be used and hence rejects this program:

```
ADSAFE.lib("Widget_", function() {
  var unusedVariable;
});
```

However, this program is perfectly safe and Untrusted-typable.

Pascal-style Declarations JSLint requires all variable declarations to be at the head of a function, so it rejects this program:

```
ADSAFE.lib("test_", function () {
  var x = 34;
  x = x + 1;
  var y = 45;
});
```

However, this program is Untrusted-typable.

Exceptions JSLint does not allow `try` blocks in untrusted programs. Exception objects leak information about the internals of ADsafe, and also contain dangerous fields with information about the stack trace at the point of throwing.

In our type system, we are conservative by necessity with respect to exceptions. Without a global analysis, it's impossible to tell which exception handlers may catch which exceptions, and any value can be thrown, so we give the exception handler the type Any. Thus, our type-checker allows `try` blocks in widgets, but any use of the *parameter* that holds the exception will be flagged as a type error, since Any is not a subtype of Untrusted. This allows widgets to catch exceptions, but prevents them from examining them.

8 Verifying the Reference Monitor

In section 7, we discussed modeling the sublanguage of widgets interacting with the sandboxing runtime. In the case of ADsafe and JSLint, we built up the Untrusted type as a specification of the kinds of values that the reference monitor, `adsafe.js`, can expect at runtime. In this section, we discuss how we use the Untrusted type to model the boundary between the reference monitor and widget code, and ensure that the runtime library correctly guards critical behavior.

8.1 ADsafe's Structure

A key property of widgets is that they do not directly interact with the Web browser environment; they use the reference monitor's "wrapper" functions instead. Figure 11 shows an outline of the structure of ADsafe and its wrapper functions. The JSLint checker allows the widget access to two special values, `dom` and `ADSAFE`, which provide the top-level entry points into the library.[15] The first provides an API for creating and searching for DOM elements, and manipulating them within the subtree allotted to the widget. The second provides protected field access and update (`ADSAFE.get` and `ADSAFE.set`), as well as several other wrapped utility functions.

The `dom` wrappers create and return `Bunch` objects, which are a simple abstraction around a list of DOM elements. They have a number of methods which provide further utilities for manipulating the DOM, and have their own set of checks for potentially malicious invocations. The widget is intended to never have direct access to the `__nodes__` field, in order to emulate that the field is private to the Bunch methods.

Figure 11 shows a number of checks that ADsafe performs:

- Checking that `this` is not the global object at the top of all Bunch methods (line 18). Since (in browsers) the global object has a field called `window` that refers back to itself, checking for the presence of that field is sufficient for

[15] This is actually a slightly simplified presentation, as there is an extra layer of instantiability to give multiple widgets on the same page different instantiations of `dom` restricted to their respective subtrees.

```
var banned = { 'arguments': true, callee: true, ... }
// other shared initialization variables...
var makeableTagName = { a: true, abbr: true, acronym: true, ... };
function error(message) { throw { name: "ADsafe", message: message }; }
function string_check(string) {
  if (typeof string !== 'string') error("ADsafe string violation.");
  return string; }
function reject_name(n) { return banned[n] || /* n has underscores*/; }
function reject_global(that) { if (that.window) { error(); } }

// ... many other helper functions

function Bunch(nodes) {
  this.__nodes__ = nodes;
}
Bunch.prototype = {
    append: function(appendage) {
      reject_global(this);
      var nodes = this.__nodes__;
      // Append things to nodes
      return this;
    },
    blur: function () { ... },
    check: function (value) { ... },
    count: function () { ... }
    // ... many Bunch methods
};
dom = {
    append: function (bunch) { ... },
    count: function () { ... },
    tag: function(name) {
      name = string_check(name);
      if(makeableTagName[name] === true) {
        return new Bunch([document.createElement(name)]);
      }
    }
    // ... many dom functions
};
ADSAFE = {
    get: function (object, name) {
        if(reject_name(name))
            error();
        else
            return object[name]
    },
    later: function (func, timeout) {
        if (typeof func === 'function')
            setTimeout(func, timeout || 0);
        else
            error();
    }
    // ...
};
```

Figure 11: The structure of `adsafe.js`

```
{
  setTimeout: Untrusted ∪ Global × (Untrusted → Untrusted) × Untrusted → Int,
  document: {
    ...
  },
  ...
}
```

Figure 12: A fragment of the type of `window`

ruling out the global object, and ensuring that an Untrusted-typed object is in fact bound to `this`.

- Checking that only appropriate types of DOM nodes are created dynamically (line 35). Notably, the `makeableTagName` whitelist doesn't contain the `"script"` tag, which allows loading new code at runtime.
- Checking that banned fields are not used, by a combination of `reject_name` and `ADSAFE.get` (lines 8 and 42).
- Checking that primitives that contain `eval`-like behavior aren't passed arguments that would exercise this (line 49).

To verify, using types, that these patterns adequately restrict behavior demands that we: define the types of the values being passed into the reference monitor (which we do with the Untrusted type); ascribe types to the Web browser's built-in APIs that capture only good behavior; and check that, given values of type Untrusted, the wrappers respect the safe types given to these APIs.

8.2 Types for the DOM

The ADsafe reference monitor directly interacts with the Web browser and the DOM. To type-check the reference monitor code, we have to ascribe types to these APIs. This appears to be daunting: the browser exposes an enormous API that is constantly growing. Furthermore, different browsers implement different APIs and even have different types for the same function name.

Fortunately, we only have to ascribe types to the fragment of these APIs that the reference monitor actually uses (500 lines of type definitions). Any function that is not typed is inaccessible to the reference monitor. (If we accidentally omit a function that ADsafe uses, our type-checker raises a type-error.)

Most of this task is routine. But, we are careful when ascribing types to a few well-known functions, such as `eval`. Since `eval` can load new code, we simply do not give it a type. This makes `eval` completely inaccessible to the reference monitor. Happily, the ADsafe reference monitor is written in a straightforward, `eval`-free manner, so this simple approach works for `eval` and related functions such as `document.write`.

```
var dom = {
  append:
  function(bunch)
  /*: Untrusted ∪ Global × Untrusted ··· → Untrusted */
    { // body of append ... },
  combine:
  function(array)
  /*: Untrusted ∪ Global × Untrusted ··· → Untrusted */
    { // body of combine... },
  q:
  function (text)
  /*: Untrusted ∪ Global × Untrusted ··· → Untrusted */
    { // body of q... },
  // ... more dom ...
};
```

Figure 13: Annotations on ADsafe's DOM wrapper

Some functions must, however, be handled more subtly. Consider the function `setTimeout`, which consumes a callback argument that it queues for invocation after the given time. We would expect this callback argument to be of function type. However, if `setTimeout` is given a *string* instead, it parses it as JavaScript and uses it as the body of the callback. This is effectively another `eval`. To prevent this backdoor, our type environment permits only function values in the callback position, as shown in fig. 12. This type environment is essentially the specification of foreign DOM functions, typically implemented in C++, that are exposed to JavaScript code.

This type environment is therefore subtle to construct. We build ours based on our and communal knowledge of unsafe functions and their arguments, and our verification is implicitly parameterized over the quality of this environment. However, as new attacks are found, we believe it is easy to modify the environment to quickly reflect the underlying vulnerabilities. Repeating the verification against this new environment would identify places where these new-found vulnerabilities affect the safety of the sandbox.

8.3 Type-Checking the Reference Monitor

The Untrusted type specifies the shape of widget values that the ADsafe runtime manipulates. Untrusted is therefore used pervasively in our verification of `adsafe.js`. A Bunch (see the constructor function `Bunch` in fig. 11) is an object with a `__nodes__` field and a number of methods for manipulating and querying `__nodes__` without directly returning the value of that field. At first glance, it appears these methods are intended to take Bunches and other flat data (like numbers and strings) as their arguments, and return flat data or new Bunches.

It is thus tempting to say that each Bunch object has the same type:

$$\mathsf{ABunch} = \left\{ \begin{array}{l} \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle, \\ \texttt{"\_\_proto\_\_"} : \left\{ \begin{array}{l} \texttt{"append"} : \mathsf{ABunch} \times \mathsf{ABunch} \to \mathsf{ABunch}, \\ \texttt{"count"} : \mathsf{ABunch} \to \mathsf{Num}, \\ \ldots \end{array} \right\} \end{array} \right\}$$

However, there are a few ways in which this simple approach is not an accurate model of of these values and their potential uses.

Bunch Values and the Untrusted Type JSLint puts no additional restrictions on the uses of Bunch values in the widget code beyond what we've already discussed, making no effort to distinguish them from other values. This program, for example, uses the `dom.tag` entry point to create a DOM node wrapped in a Bunch object, and then assigns into some of its fields:

```
var p = dom.tag("p");
p.foo = "new, exciting value";
p.append = function() { return "This function shadows Bunch.prototype.append"; };
```

A more accurate type for the value of `p` after this process is no longer the ABunch type above, but rather something like:

$$\mathsf{ABunch} = \left\{ \begin{array}{l} \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle, \\ \texttt{"foo"} : \mathsf{Str}, \\ \texttt{"append"} : () \to \mathsf{Str}, \\ \texttt{"\_\_proto\_\_"} : \left\{ \begin{array}{l} \texttt{"append"} : \mathsf{ABunch} \times \mathsf{ABunch} \to \mathsf{ABunch}, \\ \texttt{"count"} : \mathsf{ABunch} \to \mathsf{Num}, \\ \ldots \end{array} \right\} \end{array} \right\}$$

The fields `"foo"` and the shadowed `"append"` could well have been any Untrusted-typed value. If we tried to use the more restricted ABunch type on the return values of Bunch-producing methods, type-checking widgets like the above would fail, with the type-checker complaining about assignments into unknown fields. This would make our type-checking process fail to correspond with JSLint. We thus design the Untrusted type to be a *supertype* of the ABunch type, so we can *safely widen* the type of Bunch values to Untrusted. The type-checker does this when we annotate these methods with the return type Untrusted rather than ABunch, so the `"append"` field would become:

$$\texttt{"append"} : \mathsf{ABunch} \times \mathsf{ABunch} \to \mathsf{Untrusted},$$

This explains two of the components of the Untrusted type (fig. 10). The Bunch type in the `"__proto__"` field's union is an alias for the type of the Bunch prototype containing the shared methods like `"append"`. The `__nodes__` field, if defined, must contain an array of DOM nodes. These design choices are what let Bunch values be included in the Untrusted type, and thus fit into our typed account of allowed values in widget code.

One final important detail is that the type of the `"__nodes__"` field, Array⟨HTML⟩, is *not* compatible with Untrusted. Objects of type HTML have a `"__proto__"` field that is not included in the union of prototypes in Untrusted. A widget that contains a lookup on the field `"__nodes__"` will not type-check, and the type-checker will report an error that the type HTML cannot be safely widened to Untrusted. This is an interesting distinction between our type-checking approach and JSLint. JSLint bans `"__nodes__"` merely by name—as it would any other field starting with `"_"`—whereas our type-checker disallows it because of type incompatibility. Both prevent widget code from accessing the underlying unsafe objects, but for very different reasons.

Defensive Bunch Methods So far, we justified why the return type of Bunch methods should be Untrusted, and how Bunches can safely be considered Untrusted-typed values. We must next consider what the *methods* of Bunches can assume about their arguments. The type we suggested for a method like `"append"` was:

$$\mathsf{ABunch} \times \mathsf{ABunch} \to \mathsf{Untrusted}$$

With this type the method body will type-check, but we will now not be able to *invoke* this method in code that passed JSLint (a type-error will occur).

This is because JSLint allows programs to invoke methods such as `append` with invalid arguments, as long as they are Untrusted-typed. For example, the following program passes JSLint:

```
var func = someBunch.append;
func(900, true, "junk", -7);
```

After the call above, in the body of `append`, `this` is bound to `window`, the first argument is a number (not an ABunch), and there are additional arguments. We account for this behavior by weakening the type of `append`, so that it can receive the kinds of values that a widget may provide:

$$\mathsf{Untrusted} \cup \mathsf{Global} \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}$$

For `append` to do useful work, it must narrow the type of its arguments to Bunch. To do so, it applies runtime checks to ensure it does not receive invalid arguments. In this case, the checks are routine:

```
Bunch.prototype.append = function(child) {
  reject_global(this); // raises exception if Global
  var elts = child.__nodes__; // has type Array⟨HTML⟩, if the field exists
  ...
  return this;
}
```

The runtime check in `append`'s body (namely, `reject_global(this)`) is responsible for checking that `this` is not the global object before manipulating it. Our type checker recognizes such checks and narrows the broader type to Untrusted after appropriate runtime checks are applied. If such checks were missing, the type of `this` would remain Untrusted ∪ Global, and `return this` would signal a type error because Untrusted ∪ Global is not a subtype of the stated return type Untrusted.

Ascribing types to functions of the ADsafe interface is actually trivial. *All functions must have the same type*:

$$\mathsf{Untrusted} \cup \mathsf{Global} \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}$$

Therefore, most of the type annotations are trivial (fig. 13).

Types for Private Functions ADsafe also has a number of private functions that are not exposed to the widget. These functions use types that widgets cannot safely use, such as HTML. For example, ADsafe defines a `hunter` object, which contains functions that traverse the DOM and accumulate arrays of DOM nodes. These functions all have the type $\mathsf{HTML} \to \mathsf{Undef}$, and add to an array `result` that has type $\mathsf{Array}\langle\mathsf{HTML}\rangle$. ADsafe can freely use these types internally, as long as they aren't exposed to widgets. Our annotations ensure they do not, since they are not compatible with the type Untrusted.

8.4 Required Refactorings

Our type system cannot type check the ADsafe runtime as-is; we need to apply some simple refactorings. The need for these refactorings does not reflect a weakness in ADsafe. Rather, they are programming patterns that we cannot verify with our type system. Naturally, we were tempted to add new features to the type system to verify all of ADsafe as-is. However, in a few cases, we concluded that the cost of additional type-system complexity would far outweigh the trivial refactorings documented below.

Additional `reject_name` Checks ADsafe uses `reject_name` to check accesses and updates to object properties in `adsafe.js`. If-splitting uses these checks to narrow string set types and type-check object property references. However, ADsafe does not use `reject_name` in every case. For example, it uses a regular expression to parse DOM queries, and uses the result to look up object properties. Because our type system makes conservative assumptions about regular expressions, it would erroneously indicate that a blacklisted field may be accessed. Thus, we add calls to `reject_name` so the type system can prove that the accesses and assignments are safe.

We add 7 new `reject_name` checks to Bunch methods that use JavaScript objects as dictionaries. We do not believe these checks imply the presence of security bugs; they are only necessary to to pass the conservative checks in the type checker.

Inlined `reject_global` Checks Most Bunch methods begin with the assertion `reject_global(this)`, which ensures that `this` is Untrusted-typed in the rest of the method. Our type system cannot account for this non-local side-effect, because the flow analysis is intraprocedural. Instead, we inline `reject_global`, which allows if-splitting to refine types appropriately. For example, in the

`Bunch.prototype.append` example earlier in this section, we had to inline the application of `reject_global(this)`.

The inlining of `reject_global` checks has the largest impact in terms of lines of code. There are 48 Bunch methods, all of which have the check changed from a function call to an inlined `if` statement with an explicit return. This is because our type-checker can easily handle local flows [27], but cannot refine types based on calls that throw exceptions in certain conditions. Latent predicates [61] do handle interprocedural calls, but are designed for predicates rather than exceptional behavior. We believe that a type checker that combines flow typing and latent predicates would not need this refactoring.

`makeableTagName` ADsafe has a whitelist of safe DOM elements that it defines as a dictionary:

```
var makeableTagName =
  { "div": true, "p": true, "b": true, ... };
```

Notably, this dictionary omits an entry for `"script"`. The `document.createElement` DOM method creates new nodes. We ensure that `<script>` tags are not created by typing it as follows:

$$\texttt{document.createElement} : \overline{\texttt{"script"}} \rightarrow \mathsf{HTML}$$

ADsafe uses its tag whitelist before calling `document.createElement`:

```
if (makeableTagName[tagName]) {
  document.createElement(tagName);
}
```

Our type checker cannot account for this check. We instead refactor the whitelist as follows (a technique noted elsewhere [42]):

```
var makeableTagName =
  { "div": "div", "p": "p", "b": "b", ... };
```

The type of these string constants are the strings themselves. Since `"script"` is not among these string constants, the type of `makeableTagName[tagName]` now matches the argument type of `createElement`. This refactoring works because all strings in JavaScript are "truthy" values that the program can successfully branch on.

The `makeableTagName` refactoring happens in two (identical) implementations of a method named `tag`, one in the `dom` object and one on the `Bunch` prototype.

8.5 The Verification Process and Unverifiable Code

A complex body of code like the ADsafe runtime cannot be type-checked from scratch in one sitting. We therefore found it convenient to augment the type system with a `cheat` construct that ascribes a given type to an expression without descending into it. We could thus use `cheat` when we encountered a type error and wanted to make progress on a different part of the program. Our goal, of course, was to ultimately remove every `cheat` from the program.

```
var reject_name = function (name) {
    return
    ((typeof name !== 'number' || name < 0) &&
     (typeof name !== 'string' ||
      name.charAt(0) === '_' ||
      name.slice(-1) === '_' ||
      name.charAt(0) === '-'))
        || banned[name];
});

function F() {} // only used below

ADSAFE.create =
  typeof Object.create === 'function'
  ? Object.create
  : function(o) {
      F.prototype =
        typeof o === 'object' && o
               ? o : Object.prototype;
      return new F();
    };
```

Figure 14: The unverified portion of ADsafe

We were unable to remove two `cheats`, leaving eleven unverified source lines in the 1,800 LOC ADsafe runtime. We can, in fact, ascribe meaningful types to these functions, but checking them is beyond the power of our type system. Figure 14 shows these eleven unverified lines, and we discuss them below.

`reject_name` The `reject_name` function returns `true` if its argument is a blacklisted field and `false` otherwise. This function is used as a predicate to guard against invalid field accesses, so we ascribe it an intersection type:

$$(\mathsf{UnsafeField} \to \mathsf{True}) \ \cap \ (\overline{\mathsf{UnsafeField}} \to \mathsf{False})$$

where UnsafeField is the set of blacklisted field names. Our implemented typechecker does not support checking functions with intersection types, but we type-check applications of intersection-typed functions in the usual way.

`ADSAFE.create` In the ECMAScript 5 standard, `Object.create` takes an object `o` as a parameter and creates a new object whose prototype is `o`; if `o` is not an object, the new object's prototype is `Object.prototype`. ADsafe provides this same functionality for older browsers through `ADSAFE.create`. This function is never used by ADsafe; it is only intended for widgets. The implementation of `ADSAFE.create` for older browsers uses a sophisticated imperative pattern that we cannot type-check. But, by inspection, the type of `ADsafe.create` is evidently:

$$\mathsf{Global} \cup \mathsf{Untrusted} \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}$$

`ADSAFE._intercept` Finally, ADsafe allows the hosting Web page to define *interceptors*, which are functions that get direct access to the DOM. Since interceptors are application-specific and not a standard part of ADsafe, we do not bother verifying any particular set of interceptors. However, note that our tools can do better. We can verify interceptors using the same technology we use to verify the ADsafe runtime—our typechecker!

8.6 Annotation Effort

In total, we insert 322 annotations into the body of `adsafe.js` that don't change the code itself. The majority fall into three categories:

1. Untrusted-annotations on functions that are exposed to untrusted widgets,
2. upcasts on arrays and objects to the Untrusted type, and
3. upcasts on variables that are initialized after they are declared.

These annotations are straightforward to apply. For (1), the widget-facing interface of ADsafe is easy to find; it is the Bunch methods, the `dom` object, and the `ADSAFE` object. For (2), a failure to provide such an annotation results in a type error complaining about a type mismatch with Untrusted, and hence these annotations can be performed incrementally at the prompting of the typechecker. Similarly, for (3), missing such an annotation results in a type error that mentions an incompatibility between a type and Undef, since the variable was uninitialized, and the initialization point needs an upcast.

Typing the internals of ADsafe requires a deep understanding of its workings. For example, ADsafe has several functions that do the actual work of querying collections of HTML elements. These functions consume "query objects" with verify specific fields:

$$\mathsf{Selector} = \{\texttt{"op"} : \mathsf{Str}, \texttt{"name"} : \mathsf{Str}, \texttt{"value"} : \mathsf{Str}, \texttt{"\_\_proto\_\_"} : \mathsf{Object}\}$$

These fields describe operations to perform (e.g., lookup a style field, set visibility, update text value), on a set of DOM nodes. Then, a function that performs all these operations on an array of nodes has a type like:

$$\mathsf{Array}\langle\mathsf{Selector}\rangle \times \mathsf{Array}\langle\mathsf{HTML}\rangle \to \mathsf{Array}\langle\mathsf{HTML}\rangle$$

We do have to understand the internals of `adsafe.js` to add these annotations, which amounts to type-checking a mini-DOM API. This part of annotation is labor-intensive, but `adsafe.js` is remarkably easily typable. We did find some type errors during development that we reported as correctness bugs (not security vulnerabilities), because the incorrect program wouldn't type-check, or required odd-seeming types (e.g., using the same variable as a number and a string). Several of these were acknowledged and fixed by ADsafe's author.

9 ADsafety Theorems

Sections 7 and 8 gave the details of our strategy for modeling JSLint and verifying `adsafe.js`. In this section, we combine these results and relate it to the original definition of ADsafety (definition 1). The use of a type system allows us to make straightforward, type-based arguments of safety for the components of ADsafe.

The lemmas below formally reason about type-checked widgets. Claim 3 (section 7.2) establishes that linted widgets are in fact typable. Therefore, *we do not need to type-check widgets.* Widget programmers can continue to use JSLint and do not need to know about our type checker. However, given the benefits of uniformity provided by a type checker over ad hoc methods like JSLint (section 10 details one exploit that resulted from such an ad hoc approach), programmers may be well served to use our type checker instead.

Theorem 1 (ADsafety) *Given our type environment* Γ*, for all widgets* p*, if*

1. p *and all subexpressions of* p *are* Untrusted*-typable in* Γ*,*
2. *the reference monitor* r *is typable in* Γ*,*
3. *the reference monitor* r*, paired with the initial store* σ_0*, evaluates to some value* v *and the store* σ *(written* $\sigma_0, r \rightarrow^* \sigma, v$*),*
4. σ *is typable by* Γ*,*
5. $\sigma p \rightarrow \sigma' p'$ *(single-step reduction),*

then all subexpressions of p' *are* Untrusted*-typable in* Γ*, and* σ' *is typable by* Γ*.*

Proof: This follows from type-preservation [27, 51]. Note that the type system for ADsafe verification allows additional, well-known runtime errors. We argued that it is safe to do so in section 6. ■

It's useful to explain several of these pieces in more detail:

- Γ is our type environment (described in section 8.2). It has types for the built-in prototypes, the browser API, the global object, and the global identifiers that are used by both `adsafe.js` and the widget program p.
- σ is the model of the dynamic store in λ_{JS}. Its locations hold the values of objects and variables at runtime. The distinguished initial environment σ_0 represents the store at the start of computation. Each global identifier in Γ corresponds to a single location in σ_0. Each of these identifiers is substituted for its corresponding location in the store as the first step of running the program.
- We require that both `adsafe.js` and the widget program p type in the same environment Γ. This ensures that the runtime library and the untrusted program are verified with respect to the same assumptions about

the environment they run in. So, for example, the global variable `ADSAFE` created as the entry point in the reference monitor has the same type in the typing of `adsafe.js` and the widget program.

With this background, each of the preconditions of the proof warrant some justification and relation to the ADsafe system. Precondition 1 we describe in section 7, and argue via testing that all widget programs passing JSLint enjoy this typability property. Precondition 2 we justify by running our type-checker on the code of `adsafe.js`, described in section 8. Precondition 3 we will return to in a moment. For precondition 4, we assume that the type environment we write down is an accurate model of the browser runtime after `adsafe.js` has run. Precondition 5 is simply stating the execution of the program as it takes steps in λ_{JS}, modelling widget programs running in the browser.

Precondition 3 describes `adsafe.js` initialization, which is done primarily by updating global state, before the widget starts running. This update is necessary so that the widget runs in state σ rather than σ_0. We model the widget running against σ, which contains values consistent with our model of the environment, Γ. Our environment is *not* an accurate model of σ_0, since it assumes that `adsafe.js` has run (i.e., the heap in the browser is in state σ). This opens up the possibility that our verification doesn't properly account for these updates, and is missing vulnerabilities that occur as a result. In what follows, we mitigate this concern and also discuss how to guard against it.

The reference monitor makes two widget-visible changes to the environment. First, and less importantly, it assigns into the global variable `ADSAFE` with the entry points for the untrusted program to start. This is less interesting because if `adsafe.js` failed to perform this assignment, the untrusted program would simply halt with an error, so no vulnerability is masked by the presence or absence of this assignment.

Second, and more interestingly, the run-time system might provide excessive authority. For instance, in older versions of Firefox, some methods on the `Array` prototype could leak the global object. That is, for that run-time system they have a type like:

$$(\mathsf{Untrusted} \cup \mathsf{Global}) \times \mathsf{Untrusted} \cdots \to (\mathsf{Untrusted} \cup \mathsf{Global})$$

Since it would violate ADsafety to allow the untrusted program access to the global object, ADsafe assigns into the array prototype to overwrite these methods with safe versions. We type-check the wrappers that ADsafe installs at the usual safe type of

$$(\mathsf{Untrusted} \cup \mathsf{Global}) \times \mathsf{Untrusted} \cdots \to \mathsf{Untrusted}$$

This means that the wrappers themselves are safe. We ascribe these safe types to the wrapped methods in our environment, so the assignments type-check. The only thing we haven't verified is that the assignment into the `Array` prototype fields *actually happens*, overriding whatever the value was before `adsafe.js` runs. It is easy to run `adsafe.js` in a browser and see that the wrapped

functions are in fact present after it runs, giving a good deal of confidence in our model of the environment.

To fully capture this set-up behavior, we would need a type system that supports modelling strong update, capturing in the verification system information about the pre- and post-heaps. Another option would be to run `adsafe.js` in λ_{JS} and type-check the resulting store directly. For now, our tools only run over static source, and we do not account for strong update. However, as we have argued, the threat to the validity of the verification is minor, and limited to this initialization portion.

There are just two more subtleties to note about the ADsafety theorem:

- We don't require any particular type for the reference monitor, which may seem odd at first. The actual type is irrelevant because the initial communication between the reference monitor and the widget program p occurs through several shared names, and we type-check the widget program against the types of the names bound to interesting values by the reference monitor. For example, the type we give to the global variable `ADSAFE` in the widget environment assumes the type predicted (and verified) for the value bound to that name based on type-checking the runtime library. This is why the actual type of the reference monitor isn't particularly important and `adsafe.js` might as well have type Undef: it's the identifiers it binds and the structures it initializes that matter for both verification and the environment of the widget.

- We require not just that p have type Untrusted, but also that all the variables and sub-expressions of p have type Untrusted. This ensures that a program that simply happens to evaluate to a Untrusted-typed value doesn't actually get access to an unsafe value like a DOM node. For example, consider this program:

  ```
  function f() /*: → Untrusted */ {
    // document.createElement returns a direct reference to a DOM node
    var elt = document.createElement("p");
    elt.addEventListener("click", function() {
      /* Untrusted code shouldn't be able to attach events directly */ });
    return { __nodes__: [elt] }; // This *does* fit inside the Untrusted type
  }
  ```

  This function acquires a direct reference to a DOM node, which is a violation of ADsafety, despite having the safe-seeming return type of Untrusted. So, we require that all the pieces of the program have type Untrusted; this rules out access to the identifier `document`, and the return value of its call to createElement, which is a raw DOM node.

We now return to our goal of using this theorem to argue for the ADsafety properties from definition 1. To restate, this theorem says that for all widgets p whose subexpressions are Untrusted-typed, if `adsafe.js` type-checks and runs in the browser environment, p can take any number of steps and will still have the Untrusted type at all sub-expressions. Since types are preserved, two further key lemmas hold during execution:

Corollary 1 (Widgets cannot load new code at runtime) *For all widgets e, if all variables and sub-expressions of e are* Untrusted*-typed, then e does not load new code.*

Proof: By section 8.2, `eval`-like functions are not ascribed types, hence cannot be referenced by widgets or by the ADsafe runtime. Furthermore, functions that only `eval` when given strings, such as `setTimeout`, have restricted types that disallow `string`-typed arguments. Therefore, neither the widget nor the ADsafe runtime can load new code. ■

Corollary 2 (Widgets do not obtain DOM references) *For all widgets e, if all variables and sub-expressions of e are* Untrusted*-typed, then e does not obtain direct DOM references.*

Proof: The type of DOM objects is not subsumed by the Untrusted type. ■

The terse proof above is true, but we note that it is dependent on the typability of both the widget and the reference monitor. Notably, functions in the ADsafe reference monitor have the type:

$$\mathsf{Untrusted} \cup \mathsf{Global} \times \mathsf{Untrusted} \cdots \rightarrow \mathsf{Untrusted}$$

Thus, functions in the ADsafe runtime do not leak DOM references, as long as they are only applied to Untrusted-typed values. Since all subexpressions of the widget e are Untrusted-typed, all values that e passes to the ADsafe runtime are Untrusted-typed. By the same argument, e cannot directly manipulate DOM references either.

Widgets can only manipulate their DOM subtree We cannot prove this claim with our tools. JSLint enforces this property by also verifying the static HTML of widgets; it ensures that all element IDs are prefixed with the widget's ID. The wrapper for `document.getElementById` ensures that the widget ID is a prefix of the element ID. Verifying JSLint's HTML checks is beyond the scope of this work.

In addition, the wrapper for `Element.parentNode` checks to see if the current element is the root of the widget's DOM subtree. It is not clear if our type checker can express this property without further extensions.

Widgets cannot communicate This claim is false. In section 10.1, we present one counterexample and discuss others.

10 Bugs in ADsafe

Our type-checker (section 6) takes about 20 seconds to type-check `adsafe.js` (mainly due to the presence of recursive types). In some cases, type-checking failed due to the weakness of the type checker; these issues are discussed in section 8.4. The other failures, however, represent genuine errors in ADsafe

```
ADSAFE.go("AD_", function (dom, lib) {
  var myWindow, fakeNode, fakeBunch, realBunch;

  fakeNode = {
    appendChild: function(elt) {
       myWindow = elt.ownerDocument.defaultView;
    },
    tagName: "div",
    value: null
  };

  fakeBunch = {"__nodes__": [fakeNode]};

  realBunch = dom.tag("p");
  fakeBunch.value = realBunch.value;
  fakeBunch.value(""); // calls phony appendChild

  myWindow.alert("hacked");
});
```

Figure 15: Exploiting JSLint

that were present in the production system. The same applies to instances where JSLint and our typed model of it failed to conform.

In addition to the new bugs we found, we also ran our type-checker on other versions of `adsafe.js` that had vulnerabilities that were discovered before (or concurrent with) our effort. Our type-checker reported type errors on these programs too, which we describe at the end of this section.

10.1 New Bugs Found in ADsafe

By type-checking `adsafe.js` and testing our type-checker against JSLint, we found several vulnerabilities that manifested as type errors, which we report on here. All the errors listed below have been reported, acknowledged by the author, and fixed.

Missing Static Checks JSLint inadvertently allowed widgets to include underscores in quoted field names. In particular, the following expression was deemed safe:

```
fakeBunch = { "__nodes__": [ fakeNode ] };
```

A malicious widget could then create an object with an `appendChild` method, and trick the ADsafe runtime into invoking it with a direct reference to an HTML element, which is enough to obtain `window` and violate ADsafety. The full exploit is in fig. 15.

This bug manifested itself as a discrepancy between our model of JSLint as a type checker and the real JSLint. Recall from section 7 that all expressions in widgets must have type Untrusted (defined in fig. 10). For the expression

```
ADSAFE.go("AD_", function (dom, lib) {
  var called = false;
  var obj = {
    "toString": function() {
      if (called) {
        return "url(evil.xml#exp)";
      }
      else {
        called = true;
        return "dummy";
      }
    }
  };
  dom.append(dom.tag("div"));
  dom.q("div").style("MozBinding", obj);
});
```

```
<!-- evil.xml -->
<?xml version="1.0"?>
<bindings><binding id="exp">
<implementation><constructor>
document.write("hacked")
</constructor></implementation>
</binding></bindings>
```

Figure 16: Firefox-specific exploit for ADsafe

`{ "__nodes__": [fakeNode] }` to type as Untrusted, the `"__nodes__"` field must have type Array⟨HTML⟩∪Undef. However, `[fakeNode]` has type Untrusted, which signals the error.

JSLint similarly allowed `"__proto__"` and other fields to appear in widgets. We did not investigate whether they can be exploited as above, but setting them causes unanticipated behavior. Fixing JSLint was simple once our type checker found the error. (As we have said, an alternate solution would be to use our type system as a replacement for the ADsafety part of JSLint.)

Missing Runtime Checks Many functions in `adsafe.js` incorrectly assumed that they were being applied only to primitive strings. For example, `Bunch.prototype.style` began with the following check, to ensure that widgets do not programmatically load external resources via CSS:

```
Bunch.prototype.style = function(name, value) {
  if (/url/i.test(value)) { // regex match?
    error();
  }
  ...
};
```

Thus, the following widget code would signal an error:

```
someBunch.style("background",
  "url(http://evil.com/image.jpg)");
```

The bug is that if `value` is an object instead of a string, the regular-expression `test` method will inadvertently invoke `value.toString()`.

A malicious widget can construct an object with a stateful `toString` method that passes the test when first applied, and subsequently returns a malicious URL. In Firefox, we can use such an object to load an XBL resource[16] that contains arbitrary JavaScript (fig. 16).

We ascribe types to JavaScript's built-ins to prevent implicit type conversions. Therefore, we require the argument of `Regexp.test` to have type Str. However, since `Bunch.prototype.style` can be invoked by widgets, its type is Untrusted × Untrusted → Untrusted, and thus the type of `value` is Untrusted.

This bug was fixed by adding a new `string_check` function to ADsafe, which is now called in 18 functions. All these functions are not otherwise exploitable, but a missing check would cause unexpected behavior. The fixed code is typable.

Bug Bounty When ADsafe was first announced,[17] its author offered a bounty:

> If [a malicious client] produces no errors when linted with the ADsafe option, then I will buy you a plate of shrimp.

At the end of this work, the author of ADsafe bought the first two authors all seven shrimp dishes available at a restaurant. (The third author is vegetarian, illustrating the dangers of poorly-designed incentive systems.)

Counterexample to Communication Isolation Finally, a type error in the code of `Bunch.prototype.getStyle` helped us generate a counterexample to ADsafe's claim of widget noninterference (definition 1, part 4). The `getStyle` method is available to widgets, so its type must be Untrusted → Untrusted. The following code is the essence of `getStyle`:

```
Bunch.prototype.getStyle = function (name) {
  var sty;
  reject_global(this);
  sty = window.getComputedStyle(this.__nodes__);
  return sty[name];
}
```

The bug above is that `name` is unchecked, so it may index arbitrary fields, such as `__proto__`:

```
someBunch.getStyle("__proto__");
```

This gives the widget access to the prototype of the browser's `CSSStyleDeclaration` objects. Thus the return type of the body is not Untrusted, yielding a type error.

A widget cannot exploit this bug in isolation. However, it can replace built-in methods of CSS style objects and interfere with the operation of the hosting page and other widgets that manipulate styles in JavaScript.

[16] `https://developer.mozilla.org/en/XBL`

[17] `tech.groups.yahoo.com/group/caplet/message/44`

This bug was fixed by adding a `reject_name` check that is now used in this and other methods. Despite the fix, ADsafe still cannot enforce non-interference, since widgets can reference and affect properties of other shared built-ins:

```
var arr = [ ];
arr.concat.channel = "shared data";
```

The author of ADsafe pointed out the above example and retracted the claim of non-interference. (Maffeis, et al. [42] found an analogous problem in FBJS.)

Fixing Bugs and Tolerating Changes Each of our bug reports resulted in several changes to the source, which we tracked. In addition to these changes, the reference monitor also underwent non-security related refactorings during the course of this work. Even though we did not provide our type checker to its author, we easily continued type-checking the code after these changes. One change involved adding a number of new `Bunch` methods to extend the API. Keeping up-to-date was a simple task, since all the new `Bunch` methods could be quickly annotated with the Untrusted type and checked. In short, our type checker has shown robustness in the face of independent program edits.

10.2 Prior Bugs as Type Errors

Before and during our implementation, other exploits were found in ADsafe and reported [42, 58]. We ran our type checker on the exploitable code, and our tools catch the bugs and report type errors.

Misusing `this` Maffeis et al. [42] found a bug where ADsafe accidentally returned the special identifier `this` from a DOM API wrapper. The exploit involved using function invocation syntax rather than method call syntax to call the function, which causes `this` to be bound to the global object. The buggy code is (with type annotations):

```
var ephemeral;
function ephemeral_method() /*: Untrusted ∪ Global → Untrusted */ {
    if (ephemeral) {
        ephemeral.remove();
    }
    ephemeral = this;
    return this;
}
```

On this fragment, our type-checker reports an error on `return this`. The type error is simple: the `this` parameter is annotated with a union of Untrusted type with the global object type `HTMLWindow`, which does not match the annotated return type of just Untrusted.

Unrestricted Dictionary Assignment Taly et al. [58] discovered an exploit involving an unchecked dictionary assignment. This could be used to assign into the `__nodes__` field of an object and pass an object to the ADsafe runtime that could masquerade as a `Bunch` wrapper. The offending code was:

```
var adsafe_id; // a string used to detect initialization
              // not important for this bug
var adsafe_lib; // a shared dictionary for user-written
               // ADsafe libraries
function lib(name, f) /*: Untrusted × Untrusted → Untrusted */ {
    if (!adsafe_id) {
        return error();
    }
    adsafe_lib[name] = f(adsafe_lib);
}
```

The type-checker signals an error, stating that the program using a non-string expression as a field name.

Abusing `toString` An abuse of the implicit call to `toString` was also present in the check for whitelisted tag names in the wrapper for the `createElement` function of ADsafe [42]. The following code is buggy, because both dictionary lookup and the built-in `createElement` function call `toString` on their argument, so a stateful `toString` can fool the first check, and provide a value like `script` to `createElement`:

```
var makeableTagName = {"div": true, "p": true, ...};

function tag(name) /*: Untrusted → Untrusted */ {
    var node /*: upcast HTML ∪ Undef */;
    if(makeableTagName[name] === true) {
        node = document.createElement(name);
    }
}
```

The type-checker again complains that a non-string is being used in object lookup. If we had added a simple if-split, like:

```
var makeableTagName = {"div": true, "p": true, ...};

function tag(name) /*: Untrusted → Untrusted */ {
    var node /*: upcast tdom HTML ∪ Undef */;
    if(typeof name === "string") {
      if(makeableTagName[name] === true) {
          node = document.createElement(name);
      }
    } else {
      error();
    }
}
```

The type-checker would instead complain that `createElement` is passed a value outside of the restricted domain we give it, which is the set of strings in the `makeableTagNames` dictionary, as all it knows is that `name` must be a string. To the best of our knowledge, the above code actually is safe. Our type-checker simply isn't sophisticated enough to reason about it. This is why we have to refactor it to make it type-check (8.4).

11 Related Work

JavaScript Web Sandboxes ADsafe [12], BrowserShield [53], Caja [46], and FBJS [18] are archetypal Web sandboxes that use static and dynamic checks to safely host untrusted widgets. However, the semantics of JavaScript and the browser environment conspire to make JavaScript sandboxing difficult.

Maffeis et al. [40] use their JavaScript semantics to develop a miniature sandboxing system and prove it correct. Armed with the insight gained by their semantics and proofs, they find bugs in FBJS and ADsafe (which we also catch). However, they do not mechanically verify the JavaScript code in these sandboxes. They also formalize capability safety and prove that a Caja-like subset is capability safe [42]. However, they do not verify the Caja runtime or the actual Caja subset. In contrast, we verify the source code of the ADsafe runtime and account for ADsafe's static checks.

Taly, et al. [58] develop a flow analysis to find bugs in the ADsafe runtime (that we also catch). They simplify the analysis by assuming the browser runs a simpler JavaScript subset, $\text{SES}_{\text{light}}$ (Secure ECMAScript), which is augmented with a guarded property lookup operation that isn't found in real-world JavaScript. In contrast, ADsafe is designed to run on current browsers, and thus supports older and more permissive versions of JavaScript. We use λ_{JS} and associated tools (section 5), which are not limited to strict mode, and we find new bugs in the ADsafe runtime. In addition, Taly, et al. use a simplified model of JSLint. In contrast, we provide a detailed, type-theoretic account of JSLint, and also test it. We can thus find security bugs in JSLint as well.

Lightweight Self-Protecting JavaScript [44, 48] is a unique sandbox that does not transform or validate widgets. It instead solely uses reference monitors to wrap capabilities. These are modeled as security automata, but the model ignores the semantics of JavaScript. In contrast, this article and the aforementioned works are founded on detailed JavaScript semantics.

JSand [2] is another sandbox that does not require browser modification, and works by using a special JavaScript loader to load third-party code (rather than simply loading a page with `<script>` tags in the markup). Then, the loaded code is run in a sandboxed environment that is initialized by a startup script (part of Secure ECMAScript[18]). This sandboxed environment can be configured to attenuate or block access to the browser's APIs by providing guarded versions to the sandboxed code (for example, to stop the untrusted code from using `document.appendChild` to add an un-sandboxed script to the page). These guarded functions still need to be implemented correctly, since a mis-implemented guard could break the sandbox, and the work does not provide a verification. The type-based approach presented in this work may be appropriate for verifying these guards in much the same way we verify the widget-facing interface of `adsafe.js`.

Yu, et al. [63] use JavaScript sandboxing techniques to enforce various security policies on untrusted code. Their semantic model, CoreScript, simplifies the

[18] `https://code.google.com/p/es-lab/wiki/SecureEcmaScript`

DOM and scripting language. CoreScript cannot be used to mechanically verify the JavaScript implementation of a Web sandbox, which is what we present in this article.

Modeling the Web Browser There are formal models of Web browsers that are tailored to model whole-browser security properties [3, 8, 35]. These do not model JavaScript's semantics in any detail and are therefore orthogonal to semantic models of JavaScript [26, 39] that are used to reason about language-based Web sandboxes. In particular, ADsafe's stated security goals are limited to statements about JavaScript and the DOM (section 4). Therefore, we do not require a comprehensive Web-browser model.

Static and Dynamic Analysis of JavaScript GateKeeper [22] uses a combination of program analysis and runtime checks to apply and verify security policies on JavaScript widgets. GateKeeper's program analysis is designed to model more complex properties of untrusted code than we address by modeling JSLint. However, the soundness of its static analysis is proven relative to only a restricted sub-language of JavaScript, whereas λ_{JS} handles the full language. In addition, they do not demonstrate the validity of their run-time checks.

Chugh et al. [11] and VEX [6] use program analysis to detect possibly malicious information flows in JavaScript. Our type system cannot specify information flows, although we do use it to discover that ADsafe fails to enforce a desirable information flow-like property. VEX's authors acknowledge that it is unsound, and Chugh et al. do not provide a proof of soundness for their flow analysis. Our type system and analysis are proved sound.

Other static analyses for JavaScript [25, 31, 32] are not specifically designed to encode and check client-side security.

Heidegger, et al. [29] use regular expressions to describe object shapes in a manner that resembles our regular expressions. However, their invariants are dynamically checked contracts, whereas ours are statically checked types.

The F* programming language has a front-end for verifying JavaScript programs [57]. Using F*, one can specify and verify sophisticated properties that are beyond the scope of the type system that we use in this article. However, the aforementioned paper only verifies properties of small, synthetic JavaScript programs. Its verification times are also much slower than ours. It is conceivable that with more engineering effort, F* can be used to directly verify programs on the scale of ADsafe in a reasonable amount of time. Instead of pursuing that goal, the F* team has developed a fully abstract compiler from a subset of F* to JavaScript [20]. This makes it possible to write a reference monitor in F*, which has a simpler and saner semantics than JavaScript. One would still have to verify properties such as "no direct DOM references", but it would be easier to do so for F* than JavaScript. This approach entails discarding existing Web sandboxes and writing new ones, which is not the goal of our work.

Language-Based Security Schneider et al. [54] survey the design and type-based verification of language-based security systems. JavaScript Web sandboxes are inlined reference monitors [17].

Cappos, et al. [9] present a layered approach to building language sandboxes that prevents bugs in higher layers from breaking the abstractions and assurances provided by lower layers. They use this approach to build a new sandbox for Python, whereas we verify an existing, third-party JavaScript sandbox. However, our verification techniques could easily be used from the onset to build a new sandbox that is secure by construction.

Hedin and Sabelfeld [28] have developed and formalized a type system for JavaScript information flow control. Their checks occur dynamically and are thus able to tackle JavaScript features such as `eval` that our type system cannot handle. Some of this power is not necessary for ADsafe, which eschews `eval` entirely. However, we note that the claimed property of ADsafe that we falsify (section 10.1) is akin to non-interference; this means, if the property were true, our type-checker would not be able to verify it without also tackling information flow security.

IFrames IFrames are widely used for widget isolation. However, JavaScript that runs in an IFrame can still open windows, communicate with servers, and perform other operations that a Web sandbox disallows. Furthermore, inter-frame communication is difficult when desired; there are proposals to enhance IFrames to make communication easier and more secure [30]. Language-based sandboxing is somewhat orthogonal in scope, is more flexible, and does not require changes to browsers.

Runtime Security Analysis of JavaScript There are various means to secure widgets that do not employ language-based security. Some systems rely on modified browsers, additional client software, or proxy servers [13, 14, 33, 34, 38, 45, 63]. Some of these propose alternative Web programming APIs that are designed to be secure. Language-based sandboxing has the advantage of working with today's browsers and deployment methods, but our verification ideas could potentially apply to the design of some of these systems, too.

Differences from Conference Version This article is an extended version of an earlier conference paper [50]. Beyond expository improvements, it expands the technical material in several ways:

- The conference version presented some very specialized typing rules for JavaScript objects. We have since developed a generalization of that type system [51] that is more widely applicable to JavaScript programs. In this article, we give an overview of the new type system and update the presentation to use it.
- The conference version elided discussion about exactly how our type system was used in this verification. We actually relax the underlying sound-

ness theorem; this is a vital step in being able to check the ADsafe source with minimal changes. This version discusses this important detail.

- The conference paper abbreviated several technical details about ADsafe (e.g., the nature of JSLint) and our verification (e.g., the nature of λ_{JS} and its use in this work). This article has expanded discussions on these topics and it thus more self-contained than the conference version.

- Of the $1,800$ lines of JavaScript in ADsafe, our type system cannot type-check eleven. The conference version elided discussing these offending lines; this article discusses them thoroughly.

12 Conclusion

Our work has its limitations, which we summarize below. However many of these limitations are not fundamental and we believe our methodology is widely applicable to other security-critical JavaScript code.

12.1 Limitations

The work presented in this article has the following limitations:

- We require some type annotations (section 8.6) and refactorings (section 8.4) within the ADsafe reference monitor. But, these do not affect the users of ADsafe in any way, as the ADsafe interface is unchanged.

- We identify four key properties of ADsafe, but could not prove one of them (section 9). This is not an inherent limitation of a type-based strategy, as other type checkers can potentially verify these properties [24, 57].

- We do not verify eleven lines of the ADsafe reference monitor, out of $1,800$ total lines of code (section 8.5). Again, a more sophisticated type-checker might be able to type-check these.

- ADsafe checks some simple properties of untrusted CSS and HTML. Doing so is beyond the scope of our tools. We focus on checking properties of untrusted JavaScript, which is the primary concern of ADsafe also.

- We do not verify ADsafe's JSLint static checker, but build an executable, type-based model of JSLint and use tests to argue that we faithfully model JSLint (section 7.2). However, our model can be used as a JSLint replacement that is actually more flexible and equipped with safety theorems.

12.2 Beyond ADsafe

Despite these limitations, our work retrofits security guarantees onto an existing Web sandbox that was not designed for verification. This demonstrates the

power of our technique. But, naturally, our type-based strategy would be even more effective if used to drive the design of *new* sandboxes.

We propose the following concrete roadmap for sandbox designers:

1. formally specify the language of widgets using a type system;
2. use this specification to define the interface between the sandbox and untrusted code; and,
3. check that the body of the sandbox adheres to this interface using a sound type-checker.

Rather than trying to retrofit the type system's features onto existing static checks, the sandbox designer can work with the type system to guarantee safety constructively from the start. Tweaks and extensions to the type system are certainly possible—for example, one may want to design a sandboxing framework that forbids applying non-function values and looking up fields of `null`, which the current type system allows (section 9).

ADsafe shares many programming patterns with other Web sandboxes (section 3), but doesn't cover the full range of their features. We outline here some of the extensions that could be used to verify them:

Abstracting Runtime Tests Our type system accounts for inlined runtime checks, but requires some refactorings when these checks are abstracted into predicates. Larger sandboxes, like Caja, have more predicates, so refactoring them all would be infeasible. We could instead use ideas from occurrence typing [60], which accounts for user-defined predicates.

Modeling the Browser Environment ADsafe wraps a small subset of the DOM API and we manually check that this subset is appropriately typed in the initial type environment. This approach does not scale to a sandbox that wraps more of the DOM. If the type environment were instead derived from the C++ DOM implementation, we would have significantly greater confidence in our environmental assumptions.

As a step in this direction, in more recent work [36], our colleagues have generated a typed API for the DOM by parsing interface definition language specifications and generating corresponding types. This process is necessarily somewhat incomplete, but it generates a very good first approximation automatically, which can then be refined in the relevant places with manual effort.

Different Sandbox Models New sandboxes, such as SES,[19] behave somewhat differently. In the same spirit as Lightweight Self-Protecting JavaScript, the reference monitor, like `adsafe.js`, is a JavaScript program that is loaded before widgets. Upon loading, the program walks the object graph of the browser

[19] `https://code.google.com/p/es-lab/wiki/SecureEcmaScript`

runtime, starting at globally available references, and either removes or interposes on all potentially dangerous operations. This approach, as opposed to the sandbox architecture described in this paper (which involves static checks and a "passive" reference monitor), has the virtue that widget programs do not first need to pass through an external checker. In fact, after setting up the initial safe environment, SES itself fearlessly calls an `eval`-like function on widget code to start it! This reduces the friction in the build and execution processes and better integrates sandboxing into existing JavaScript toolchains.

These sandboxing approaches rely to some extent on newer versions of JavaScript (ECMAScript 5 and later), which seal off many of the attack surfaces that systems like ADsafe had to address. For example, ADsafe prevents access to the `constructor` field across all objects, since it gives access to the built-in `Function` object, which has `eval`-like behavior. In contrast, SES uses ES5's *getters* to execute functions at property access time. This allows SES to interpose on accesses to `Function` and provide a safe wrapped version, which has a benign `constructor` field. Thus untrusted code is free to use the `constructor` field of all objects, with SES ensuring that only safe versions are ever seen. While we would still need to verify the source of the reference monitor, we would no longer need to encode a rich static invariant about widgets: this invariant would instead be implicit in the semantics of the language and in the types of the DOM. Our semantics for ES5 [49], which like λ_{JS} has also been tested for conformance, is likely to be valuable here.

Acknowledgments

We thank Spiridon Eliopoulos for his contributions to a previous version of this work; Douglas Crockford for discussions, open-mindedness, insightful feedback, and several plates of shrimp; Mark S. Miller, Sergio Maffeis, Ankur Taly, and John Mitchell for enlightening discussions; Matthias Felleisen, Andrew Ferguson, and David Wagner for numerous comments that helped us understand weaknesses in our exposition; our anonymous reviewers for the kind of detailed and thorough reading that only the journal process provides, and which substantially improved this presentation; Andrei Sabelfeld for editorial shepherding; the NSF and Google for financial support; StackOverflow and Claudiu Saftoiu (our lower-latency version of StackOverflow) for unflagging attention to detail; and Schloss Dagstuhl, for putting social networking over computer networking.

References

[1] Microsoft Web Sandbox, 2012. `http://websandbox.livelabs.com`.

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Annual Computer Security Applications Conference*, 2012.

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of Web security. In *IEEE Computer Security Foundations Symposium*, 2010.

[4] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Handscom Field, Bedford, Massachusetts 01730, October 1972.

[5] Ihab Awad, Tyler Close, Adrienne Felt, Collin Jackson, Ben Laurie, Felix Lee, Ka-Ping Lee, David-Sarah Hopwood, Jasvir Nagra, Eric Sachs, Mike Samuel, Mike Stay, and David Wagner. Caja external security review. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/Caja_External_Security_Review_v2.pdf`.

[6] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, 2010.

[7] Armorize Malware Blog. "HDD Plus" malware spread through major ad networks, using malvertising and drive-by download, 2009. `http://blog.armorize.com/2010/12/hdd-plus-malware-spread-through.html`.

[8] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: Formalizing the core of a Web browser. In *USENIX Conference on Web Application Development*, 2010.

[9] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *ACM Conference on Computer and Communications Security*, 2010.

[10] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2012.

[11] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[12] Douglas Crockford. ADSafe. `www.adsafe.org`, 2011.

[13] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sanboxing JavaScript to fight malicious websites. In *Symposium On Applied Computing*, 2010.

[14] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference*, 2009.

[15] ECMAScript language specification, 1999.

[16] ECMAScript language specification, 2009.

[17] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[18] Facebook. FBJS, 2011. `http://developers.facebook.com/docs/fbjs/`.

[19] Matthew Finifter, Joel Howard Willis Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Network and Distributed System Security Symposium*, 2010.

[20] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yvew Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[21] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a program logic for JavaScript. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[22] Salvatore Guarnieri and Benjamin Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

[23] Arjun Guha. *Semantics and Types for Safe Web Programming*. PhD thesis, Brown University, 2012.

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

[25] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *International World Wide Web Conference*, 2009.

[26] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[27] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, 2011.

[28] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *IEEE Computer Security Foundations Symposium*, 2012.

[29] Philip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[30] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for Web mashups. In *International World Wide Web Conference*, 2007.

[31] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, 2009.

[32] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, 2010.

[33] Trevor Jim, Nikhil Swamy, and Michael Hicks. BEEP: Browser-enforced embedded policies. In *International World Wide Web Conference*, 2007.

[34] Emre Kıcıman and Benjamin Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Symposium on Operating System Principles*, 2007.

[35] Benjamin S. Lerner, Matthew J. Carroll, Dan P. Kimmel, Hannah Quayde la Vallee, and Shriram Krishnamurthi. Modeling and reasoning about DOM events. In *USENIX Conference on Web Application Development*, 2012.

[36] Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security*, 2013.

[37] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting type systems for JavaScript. In *ACM SIGPLAN Dynamic Languages Symposium*, 2013.

[38] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium*, 2010.

[39] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[40] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security*, 2009.

[41] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Run-time enforcement of secure JavaScript subsets. In *Web 2.0 Security and Privacy*, 2009.

[42] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE Symposium on Security and Privacy*, 2010.

[43] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted JavaScript. In *IEEE Computer Security Foundations Symposium*, 2009.

[44] Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting JavaScript. In *OWASP AppSec Research*, 2010.

[45] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[46] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[47] Mozilla Developer Network. Firefox OS, 2013. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS`.

[48] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[49] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *ACM SIGPLAN Dynamic Languages Symposium*, 2012.

[50] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[51] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and types for objects with first-class member names. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2012.

[52] Joe Gibbs Politz, Hannah Quay-de la Vallee, and Shriram Krishnamurthi. Progressive types. In *ACM International Symposium On New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2012.

[53] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Symposium on Operating Systems Design and Implementation*, 2006.

[54] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[55] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[56] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *European Symposium on Programming*, 2009.

[57] Nikhil Swamy, Joel H. W. Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[58] Ankur Taly, Úlfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

[59] The New York Times. On the web, ads can be a security hole, 2009. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[60] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[61] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[62] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

[63] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[64] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the Web. In *International World Wide Web Conference*, 2009.