ADsafety
Type-Based Verification of JavaScript Sandboxing

Joe Gibbs Politz Spiridon Aristides Eliopoulos Arjun Guha Shriram Krishnamurthi

Brown University

Abstract

Web sites routinely incorporate JavaScript programs from several sources into a single page. These sources must be protected from one another, which requires robust sandboxing. The many entry-points of sandboxes and the subtleties of JavaScript demand robust verification of the actual sandbox source. We use a novel type system for JavaScript to encode and verify sandboxing properties. The resulting verifier is lightweight and efficient, and operates on actual source. We demonstrate the effectiveness of our technique by applying it to ADsafe, which revealed several bugs and other weaknesses.

1 Introduction

A *mashup* Web page displays content and executes JavaScript from various untrusted sources. Facebook applications, gadgets on the iGoogle homepage, and various embedded maps are the most prominent examples. By now, mashups have become ubiquitous. Indeed, web pages that display advertisements from ad networks are also mashups, because they often employ JavaScript for animations and interactivity. A survey of popular pages shows that a large percentage of them include scripts from a diverse array of external sources [41]. Unfortunately, these third-party scripts run with the same privileges as trusted, first-party code served directly from the originating site. Hence, the trusted site is susceptible to attacks by maliciously crafted third-party software.

This paper addresses language-based Web sandboxing systems, one of several mechanisms for securing mashups. Most sandboxing mechanisms have similar high-level goals and designs, which we outline in section 2. In section 3, we review the design and implementation of sandboxes and demonstrate the need for tool-supported verification. Section 4 provides a detailed plan for the rest of the paper. Our work makes several contributions:

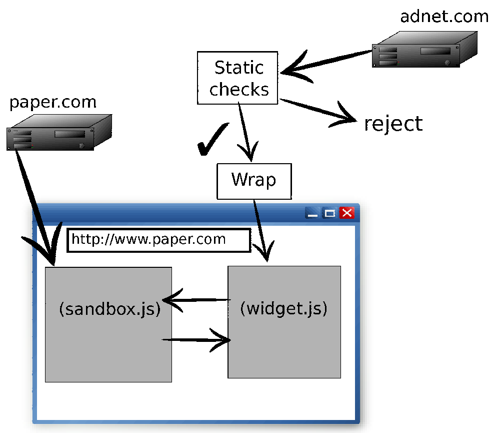

Figure 1: Web sandboxing architecture

1. A type system for general JavaScript programs, with support for patterns found in sandboxing libraries;[1]
2. a formal definition of safety properties for Yahoo!'s ADsafe sandbox in terms of this type system; and,
3. a type-based verification of the ADsafe framework, and descriptions of bugs and their fixes found while performing the verification.

2 Language-based Web Sandboxing

The Web browser environment provides references to objects that implement network access, disk storage, geolocation, and other capabilities. Legitimate web applications use them for various reasons, but embedded widgets can exploit them because all JavaScript on a page runs in the same global environment. A Web sandbox thus attenuates or prevents access to these capabilities, allowing pages to safely embed untrusted wid-

[1]See `cs.brown.edu/research/plt/dl/adsafety/v1` for our implementation, proofs, and other details.

gets. ADsafe [9], Caja [33], FBJS [13], and Browser-Shield [35] are *language-based* sandboxes that employ broadly similar security mechanisms, as defined by Maffeis, et al. [27]:

- 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 is delivered to the browser.
- A Web sandbox provides runtime *wrappers* that attenuate access to the DOM and other capabilities. 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. Web sandboxes thus specify how potentially-unsafe programs are *rewritten* to use dynamic safety checks.

This architecture is illustrated in figure 1, where an untrusted widget from `adnet.com` is embedded in a page from `paper.com`. The untrusted widget is filtered by the static checker. If static checking passes, the widget is rewritten to invoke the 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 A Web sandbox implements a *reference monitor* between the untrusted widget and the browser's capabilities. Anderson's seminal work on reference monitors identifies their certification demands [3, p 10-11]:

> The proof of [a reference monitor's] model 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.

3 Code-Reviewing Web Sandboxes

Imagine we are confronted with a Web sandbox and asked to ascertain its quality. One technique we might employ is a code-review. Therefore, we perform an imaginary review of a Web sandbox, focusing on the details of ADsafe. Later, we will discuss how to (mostly) remove people from the loop.

ADsafe, like all Web sandboxes, consists of two interdependent components:

- a static verifier, called JSLint,[2] which filters out widgets not in a safe subset of JavaScript, and
- a runtime library, `adsafe.js`, which implements DOM wrappers and other runtime checks.

These conspire to make it safe to embed untrusted widgets, though "safe" is not precisely defined. We will return to the definition of safety in section 4.

Attenuated Capabilities Widgets should not be able to directly reference various capabilities in the browser environment. Direct DOM references are particularly dangerous because, from an arbitrary DOM reference, `elt`, a widget can simply traverse the object graph and obtain references to all capabilities:

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

Widgets therefore manipulate *wrapped* DOM elements instead of direct references. DOM wrappers form the bulk of the runtime library and include many dynamic checks and patterns that need to be verified:

- The runtime manipulates DOM references, but returns them to the widget in wrappers. We must verify that all returned values are in fact wrapped, and that the runtime cannot be tricked into returning a direct DOM reference.
- The runtime calls DOM methods on behalf of the widget. Many methods, such as `appendChild` and `removeChild`, require direct 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 runtime attaches DOM callbacks on behalf of the widget. These callbacks are invoked by the browser with event arguments that include direct DOM references. We must verify that the runtime appropriately wraps calls to untrusted callbacks in the widget.
- The widget has access to a DOM subtree that it is allowed to manipulate. The runtime ensures that the widget only manipulates elements in this subtree. We must verify that various DOM traversal methods, such as `document.getElementById` and `Element.getParent`, do not allow the widget obtain wrappers to elements outside its subtree.
- The runtime wraps many DOM functions that are only conditionally safe. For example, `document.createElement` is usually safe, unless it

[2] JSLint can perform other checks that are not related to ADsafe. In this paper, "JSLint" refers to JSLint with ADsafe checks enabled.

is used to create a `<script>` tag, which can load arbitrary code. Similarly, the runtime may 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.

ADsafe's DOM wrappers are called *Bunches*, which wrap collections of HTML elements. There are twenty Bunch-manipulating functions that are exposed to the widget—in addition to several private helper functions—that face all the issues enumerated above and need to be verified. These functions cannot be verified in isolation, because their correctness is dependent on assumptions about the kinds of values they receive from widgets. These assumptions are discharged by the static checks in JSLint and other runtime checks to avoid loopholes and complexities in JavaScript's semantics.

JavaScript Semantics A Web sandbox must contend with JavaScript features that hinder security:

- Certain JavaScript features are unsafe to use in widgets. For example, a widget can use **`this`** to obtain `window`, so it is rejected by JSLint:

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

  We must verify that the subset of JavaScript admitted by the static checker does not violate the assumptions of the runtime library.
- Many JavaScript operators and functions include implicit type conversions and method calls that are difficult to reason about. For example, when an operator expects a string but is instead given an object, it does not signal an error. Instead, it calls the object's `toString` method. It is easy to write a stateful `toString` method that returns different strings on different calls. Such an object can then circumvent dynamic safety checks that are not carefully written to avoid triggering implicit method calls. These implicit calls are avoided by carefully testing the runtime types of untrusted values, using the **`typeof`** operator. Such tests are pervasive in ADsafe. As a further precaution, ADsafe tries to ensure that widgets cannot define `toString` and `valueOf` fields in objects.

JavaScript Encapsulation JavaScript objects have 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 various other dangerous fields that are also *blacklisted* and hence rejected by sandboxes. However,

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 2: Similar Rewritings for `obj[name]`

syntactic checks alone cannot determine whether computed field names are unsafe:

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

Widgets are instead rewritten to use runtime checks that restrict access to these fields. Figure 2 shows the rewrites employed by various sandboxes. Some sandboxes insert these and other checks automatically, giving the illusion of programming in ordinary JavaScript; ADsafe is more spartan, requiring widget authors to insert the dynamic checks themselves; but the principle remains the same.

Web sandboxes also simulate private fields with this method by introducing fields and then preventing widgets from accessing them. 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 paper presents a few more (section 9). 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 [4]. 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 so that readers can both determine whether these fit their needs and check whether these are implemented correctly.

4 Verifying a Sandbox: Our 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 automate the review process. However, before we begin automating anything, we need some definition of what security means. We focus on a definition that is specific to ADsafe, though the properties are similar to the goals of other web sandboxes. From correspondence with ADsafe's author, we initially obtained the following list of intended properties (rewritten slightly to use the terminology of this paper).

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

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.*

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 by preventing direct access.

We also note that the assumption about built-in prototypes is often violated in practice [14]. Nevertheless, like ADsafe, we make this assumption; mitigating it is outside our scope. Given this definition, our goal is to produce a (mostly) automated verification that supports these properties.

Verifying Safety In this paper we perform this automation using static types, presenting a type-based approach for defining and verifying the invariants of ADsafe. While one could build a custom tool to do this, we are able to perform our verification by extending (as discussed in section 11) a type checker [18] intended for traditional type-checking of JavaScript.

We choose a static type system as our tool of choice for several reasons. Programmers are familiar with type systems, and ours is mostly standard (we discuss non-standard features in sections 5 and 7). This lessens the burden on sandbox developers who need to understand what the verification is saying about their code. Second, our type system is much more efficient than most whole-program analyses or model checkers, leading to a quick procedure for checking ADsafe's runtime library (20 seconds). Efficency and understandability allow for incremental use in a tight development loop. Finally, our type system is accompanied by a soundness proof. This property accomplishes 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 order to demonstrate the effectiveness of our type-based verification approach, we use type-based arguments to prove ADsafety. We mostly achieve this (section 8) after fixing bugs exposed by our type checker (section 9). The rest of this paper presents a typed account of untrusted widgets and the ADsafe runtime.

- The ADsafety claim is predicated on widgets passing the JSLint checker. Therefore, we need to model JSLint's restrictions. We do this in section 5.
- Once we know what we can expect from JSLint, we can verify the actual reference monitor code in `adsafe.js` using type-checking (section 7).
- Before we can verify `adsafe.js`, we need to account for the details of JavaScript source and model the browser environment in which this code runs. Section 6 presents this additional work.

We discuss extensions to verify other Web sandboxes in section 10.

5 Modeling Secure Sublanguages

All web sandboxes' runtime libraries expect to execute against widgets that have been statically checked and rewritten, as shown in figure 1. These checks and rewrites enforce that widgets are written in a sublanguage of JavaScript. This sublanguage ought to be specified explicitly. We focus here on modeling the checks performed by JSLint, ADsafe's static checker, which presents an interesting challenge: there is no formal specification of the language of JavaScript programs that pass JSLint. Instead, the specification is implicit in the implementation of JSLint itself. In this section, we design a specification for JSLint-ed widgets and give confidence in its correctness.[3]

Only a fraction of JSLint's static checks are related to ADsafe. The rest are `lint`-like code-quality checks. JSLint also checks the static HTML of a widget. Verifying this static HTML is beyond the scope of our work; we do not discuss it further. We instead focus on the security-critical static JavaScript checks in JSLint.

[3]Because we want a strategy that extends to other sandboxes, we do not try to exploit the fact that JSLint is written in JavaScript. The Cajoler of Caja is instead written in Java, and the filters and rewriters for other sandboxes might be written in other languages. The strategy we outline here avoids both getting bogged down in the details of all these languages as well as over-reliance on JavaScript itself.

$$
\begin{array}{rcl}
\alpha & := & \text{type identifiers} \\
T & := & \mathsf{Num} \mid \mathsf{Str} \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Undef} \mid \mathsf{Null} \\
& \mid & \mathsf{Ref}\, T \mid \forall\alpha.T \mid \mu\alpha.T \\
& \mid & [T]T \times \ldots \times T \times T \cdots \to T \\
& \mid & \top \mid \bot \mid T \cup T \mid T \cap T \mid \mathsf{Array}\langle \mathrm{T} \rangle \\
& \mid & \{\star : F, proto : T, code : T, f : F, \ldots\} \\
& \mid & (f, \ldots)^{+} \mid (f, \ldots)^{-} \\
F & := & T \mid ☠ \mid \mathsf{Absent}
\end{array}
$$

Figure 3: Type Language for ADsafe and Widgets

How is JSLint used? The ADsafe runtime makes several assumptions about the shape of values it receives from widgets. These assumptions are not documented precisely, but they correspond to various static checks in JSLint. To model JSLint, we reflect these checks in a *type*, called Widget, which we define below. In section 5.2 we discuss how this type relates to the behavior of the JSLint implementation.

5.1 Defining Widget

We expect that *all variables and sub-expressions* of widgets are typable as Widget. The ADsafe runtime can thus assume that widgets only manipulate Widget-typed values. Our full type language is shown in figure 3 and introduced gradually in the rest of this section.

Primitives JSLint admits JavaScript's primitive values, with trivial types:

$$
\begin{array}{rcl}
\mathsf{Prim} & = & \mathsf{Num} \cup \mathsf{Str} \cup \mathsf{True} \cup \mathsf{False} \\
& & \cup \mathsf{Null} \cup \mathsf{Undef}
\end{array}
$$

We have separate types for True and False because they are necessary to type-check `adsafe.js` (section 7). Prim is an untagged union type, and our type system accounts for common JavaScript patterns for discriminating unions. We might initially assume that

$$\mathsf{Widget} = \mathsf{Prim}$$

Objects and Blacklisted Fields JSLint admits object literals but blacklists certain field names as dangerous. All other fields are allowed to contain widget values. We therefore augment the Widget type to include objects. An object type explicitly lists the names and types of various fields in an object. In addition, the special field $\star$ specifies the type of all other fields:

$$
\mathsf{Widget} = \mu\alpha.\mathsf{Prim} \cup \mathsf{Ref} \left\{
\begin{array}{l}
\star : \alpha, \\
\texttt{"arguments"} : ☠, \\
\texttt{"caller"} : ☠, \\
\texttt{"callee"} : ☠, \\
\texttt{"eval"} : ☠, \\
\ldots \\
\texttt{"toString"} : \mathsf{Absent}, \\
\texttt{"valueOf"} : \mathsf{Absent}
\end{array}
\right\}
$$

The full list of blacklisted fields is in figure 4. Our type checker signals a type error on any ☠-typed field access or assignment. This mirrors the behavior of JSLint, which also rejects field accesses and assignments on blacklisted fields (e.g., `o["constructor"]` is rejected by both the type checker and JSLint).

The Ref tag indicates that the object is mutable. We use a recursive type (μ) to indicate that all other fields, $\star$, may recursively contain Widget-typed values.[4] JSLint tries to ensure that objects in widgets do not have `toString` and `valueOf` properties. We model this with a type Absent, which ensures these fields are not present.

Absent and ☠ properties are subtly different. ☠ models fields that are intended to be inaccessible, and hence looking them up is untypable. In contrast, the typing rule for Absent field lookup performs the lookup with the type of the *proto* field, which we introduce below. Section 7.1 contains the details of type-checking field access.

Functions Widgets can create and apply functions, so we must widen our Widget type to admit them. Functions in JavaScript are objects with an internal *code* field, which we add to allowed objects:

$$
\ldots \mathsf{Ref} \left\{
\begin{array}{l}
code : [\mathsf{Global} \cup \alpha]\alpha \cdots \to \alpha, \\
\star : \alpha, \\
\ldots
\end{array}
\right\}
$$

The type of the *code* field indicates that widget-functions may have an arbitrary number of Widget-typed arguments and return Widget-typed results.[5] It also specifies that the type of the implicit **`this`**-argument (written inside brackets) may be either Widget or Global. The type Global is not a subtype of Widget, which expresses the underlying reason for JSLint's rejection of all widgets that contain **`this`** (see Claim 1 below). If the **`this`**-annotation is omitted, the type of **`this`** is $\top$.

Prototypes JSLint does not allow widgets to explicitly manipulate objects' prototypes. However, since field

[4] $\mu\alpha.T$ binds the type variable α in the type T to the whole type, $\mu\alpha.T$. Therefore, α is in fact the type Widget.

[5] The $\alpha \cdots$ syntax is a literal part of the type, and means the function can be applied to any number of additional α-typed arguments.

lookup in JavaScript implicitly accesses the prototypes, we specify the type of prototypes in Widget:

$$\ldots\,\mathsf{Ref}\left\{\begin{array}{l} \mathit{proto} : \mathsf{Object} \cup \mathsf{Function} \cup \ldots, \\ \star : \alpha, \\ \ldots \end{array}\right\}$$

The *proto* field enumerates several safe prototypes, but notably omits DOM prototypes such as `HTMLElement`, since widgets should not obtain direct references to the DOM.

Typing Private Fields In addition to explicitly blacklisted field names, JSLint also blacklists all field names that start and end with an underscore. This effectively blacklists the `__proto__` field, which gives direct access to the prototype-chain, and the `__nodes__` and `__star__` fields, which `adsafe.js` uses internally to build the Bunch abstraction. To keep our types simple, we enumerate these three fields instead of pattern-matching on field names:

$$\ldots\,\mathsf{Ref}\left\{\begin{array}{l} \texttt{"\_\_\_nodes\_\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_proto\_\_"} : \text{☠}, \\ \texttt{"\_\_\_star\_\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \star : \alpha, \\ \ldots \end{array}\right\}$$

The `__proto__` field is ☠-typed, like other blacklisted fields that are never used. However, the ADsafe runtime uses `__nodes__` and `__star__` as private fields. The types specify that ADsafe stores DOM references in the `__nodes__` field.

The full Widget type in figure 4 is a formal specification of the shape of values that `adsafe.js` receives from and sends to widgets. This type is central to our verification of `adsafe.js` and of JSLint.

5.2 Widget and JSLint Correspondence

Though we have offered intuitive arguments for why Widget corresponds to the checks in JSLint, we would like to gain confidence in its correspondence with the behavior of the actual JSLint program that sites use:

Claim 1 (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 Widget-typed.*

We validate this claim by testing. 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 should be untypable and unlintable). Note the direction of the implication: an unlintable widget may still be typable, since our type checker admits safe widgets that JSLint rejects.[6] The type checker could be used as a replacement for JSLint's ADsafe checks, but these tests give us confidence that checking the Widget type corresponds to what JSLint admits in practice.

$$\mathsf{Widget} = \mu\alpha.$$

$$\mathsf{Str} \cup \mathsf{Num} \cup \mathsf{Null} \cup \mathsf{Bool} \cup \mathsf{Undef} \cup$$
$$\mathsf{Ref}\left\{\begin{array}{l} \mathit{proto} : \begin{array}{l} \quad\mathsf{Object} \cup \mathsf{Function} \\ \cup\mathsf{Bunch} \cup \mathsf{Array} \cup \mathsf{RegExp} \\ \cup\mathsf{String} \cup \mathsf{Number} \cup \mathsf{Boolean}, \end{array} \\ \star : \alpha, \\ \mathit{code} : [\mathsf{Global} \cup \alpha]\alpha \cdots \to \alpha, \\ \texttt{"\_\_\_nodes\_\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_\_star\_\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \texttt{"caller"} : \text{☠},\ \texttt{"callee"} : \text{☠}, \\ \texttt{"eval"} : \text{☠}, \texttt{"prototype"} : \text{☠}, \\ \texttt{"watch"} : \text{☠}, \texttt{"constructor"} : \text{☠}, \\ \texttt{"\_\_proto\_\_"} : \text{☠},\ \texttt{"unwatch"} : \text{☠}, \\ \texttt{"arguments"} : \text{☠},\ \texttt{"valueOf"} : \mathsf{Absent}, \\ \texttt{"toString"} : \mathsf{Absent} \end{array}\right\}$$

Figure 4: The Widget type

6 Modeling JavaScript and the Browser

Verification of a Web sandbox must account for the idiosyncrasies of JavaScript. It also needs to model the runtime environment—provided by the browser—in which the sandboxed code will execute. Here we discuss how we model the language and the browser.

JavaScript Semantics We use the semantics of Guha, et al. [17], which reduces JavaScript to a core semantics called λ_{JS}. This latter language models the "essentials" of JavaScript: prototype-based objects, first-class functions, basic control operators, and mutation.

λ_{JS} thus omits many of JavaScript's complexities, but it is accompanied by a *desugaring* function that maps all JavaScript programs (idiosyncrasies included) to behaviorally equivalent λ_{JS} programs. The transformation explicates much of JavaScript's implicit semantics. Hence, we find it easier to build tools that analyze the much smaller λ_{JS} language than to directly process JavaScript.

Does desugaring faithfully map JavaScript to λ_{JS}? Guha, et al. test their desugaring and semantics on portions of the Mozilla JavaScript test suite. On these tests, λ_{JS} programs produce exactly the same output as JavaScript implementations. Hence, their work substantiates the following two claims.

[6]The supplemental material contains examples of the differences.

```
{
  eval: ☠,
  setTimeout: (Widget → Widget) × Widget → Int,
  document: {
    write: ☠,
    writeln: ☠,
    ...
  },
  ...
}
```

Figure 5: A Fragment of the Type of `window`

Claim 2 (Desugaring is Total) *For all JavaScript programs e, desugar$[\![e]\!]$ is defined.*

Claim 3 (Desugar Commutes with Eval) *For all JavaScript programs e, desugar$[\![eval_{JavaScript}(e)]\!] = eval_{\lambda_{JS}}(desugar[\![e]\!])$.*

This testing strategy, and the simplicity of implementation that λ_{JS} enables, give us confidence that our tools correctly account for JavaScript.

Modeling the Browser DOM ADsafety claims that `window.eval` is not applied. To validate this claim, we mark `eval` with ☠ from section 5, which marks banned fields. There are many `eval`-like function in Web browsers, such as `document.write`; these are also marked ☠. Finally, certain functions, such as `setTimeout`, behave like `eval` when given strings as arguments. ADsafe does need to call these functions, but it is careful to never call them with strings. In our type environment, we give them restrictive types that disallow string arguments.

Figure 5 specifies a fragment of the type of `window`, which carefully specifies the type of unsafe functions in the environment. The remaining safe DOM does not need to be fully specified. `adsafe.js` only uses a small subset of the DOM methods. These methods require types. The browser environment is therefore modeled with 500 lines of object types (one field per line). This type environment is essentially the specification of foreign DOM functions imported into JavaScript.

7 Verifying the Reference Monitor

In section 5, we discussed modeling the sublanguage of widgets interacting with the sandboxing runtime. In the case of ADsafe and JSLint, we built up the Widget 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 Widget type to model the boundary between reference monitor and widget code,

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

Figure 6: Annotations on the `dom` object

and ensure that the runtime library correctly guards critical behavior.

The Widget type specifies the shape of widget values that the ADsafe runtime manipulates. Widget is therefore used pervasively in our verification of `adsafe.js`. For example, consider a typical Bunch method:

```
Bunch.prototype.append = function(child) {
  reject_global(this);
  var elts = child.__nodes__;
  ...
  return this;
}
```

The Bunch objects that ADsafe passes to the widget have `Bunch.prototype` as their *proto* (see figure 4), making these methods accessible. Their use in the widget is constrained only by JSLint, so we must type-check these methods with (only) JSLint's assumptions in mind.

For example, we might assume that the `child` argument above should be a Bunch, the implicit **`this`** argument should also be a Bunch, and it therefore returns a Bunch. However, JSLint does not provide such strong guarantees. Consider this example, which passes JSLint:

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

Here, **`this`** is bound to `window`, `child` is a number, and there are additional arguments. Therefore, we cannot assume that `append` has the type $[\mathsf{Bunch}]\mathsf{Bunch} \rightarrow \mathsf{Bunch}$. Instead, the most precise type we can ascribe is:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

That is, **`this`** could be Widget-typed or the type of the global object, Global, and the other arguments may have any subtype of Widget, which includes strings, numbers, and other non-Bunch types. 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 Widget after appropriate runtime checks are applied (section 7.1). If such checks were missing, the type of **`this`** would remain Widget ∪ Global, and **`return this`** would signal a type error because Widget∪Global is not a subtype of the stated return type Widget.

Ascribing types to functions provided by the ADsafe runtime is therefore trivial. We give all the same type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \to \mathsf{Widget}$$

The type checker we extend is not ADsafe-specific, and requires explicit type annotations. However, since all the annotations are identical, they are trivial to insert. Figure 6 shows a small excerpt of such annotations, which the checker reads from comments, so programs can run unaltered in the browser.

Types for Private Functions ADsafe also has a number of private functions, which are not exposed to the widget. These functions have types with capabilities the widget does not have access to, such as HTML. For example, ADsafe specifies a `hunter` object, which contains functions that traverse the DOM and accumulate arrays of DOM nodes. These functions all have the type HTML $\to$ Undef, and add to an array `result` that has type Array⟨HTML⟩. ADsafe can freely use these capabilities inside the library as long as it doesn't hand them over to the widget. Our annotations show that it doesn't, because these types are not compatible with Widget.

7.1 Type System Highlights

In section 5 and 6, we presented types for safe objects and for values in the browser environment. We build upon earlier work on type systems that has been applied to JavaScript [18]. In this section, we present the non-standard portions of our type system that we use for typing operations on objects, sensitive conditionals, and some idiosyncrasies of JSLint and `adsafe.js`.

Object Properties and String Set Types In JavaScript, object properties (or "fields") are merely string indices: even `o.x` is just an alias for `o["x"]`. In addition, these strings can be computed and flow through the program before they are used to look up fields. Sandboxes thus deal with whitelists and blacklists of property names. To model this, we enrich the type language with sets of strings. For example, $(\texttt{"\_\_\_nodes\_\_\_"}, \texttt{"\_\_proto\_\_"})^-$ is the type of all strings *except* `"___nodes___"` and `"__proto__"`, and $(\texttt{"x"}, \texttt{"foo"})^+$ is the type of exactly `"x"` and `"foo"`.

Figure 7 shows typing rules and operations for string sets. Sets support combination via unions, subtyping via adding new strings, and subtyping of positive and negative sets. Both kinds of string sets can also be promoted to the common supertype of Str, which is equivalent to the negative string set with no entries.

$$\textsc{T-StringSet} \quad \Sigma; \Gamma \vdash str : (str)^+$$

$$\textsc{ST-StringSet}^+ \quad \frac{\forall f \in (f_1, \ldots), f \in (s_1, \ldots)}{(f_1, \ldots)^+ <: (s_1, \ldots)^+}$$

$$\textsc{ST-StringSet}^- \quad \frac{\forall f \in (f_1, \ldots), f \notin (s_1, \ldots)}{(f_1, \ldots)^+ <: (s_1, \ldots)^-}$$

$$\textsc{ST-String}^+ \quad (f_1, \ldots)^+ <: \mathsf{Str}$$

$$\textsc{ST-String}^- \quad (f_1, \ldots)^- <: \mathsf{Str}$$

$$\textsc{Equiv-Str} \quad \mathsf{Str} <: ()^-$$

Figure 7: Typing and operations on string set types

Equipped with string sets, we can describe the typing of object property dereference. When the property name is a string set, we union the types of the properties that are members of the string set, paying careful attention to absent fields and prototype lookup. Figure 8 shows the rule T-LOOKUP, with examples shown in figure 9.

String sets allow the type checker to avoid certain named properties, as in the last example of figure 9, where the `"eval"` property has the bad type ☠ but the string set type of the index excludes `"eval"`. The rule for property update (not shown here) is similar but simpler, as property update in JavaScript does not recur inside prototypes, and only operates on the property names of the top-level object.

If-Splitting A reference monitor has various runtime checks to ensure that protected objects—DOM objects and browser functions in ADsafe's case—are only manipulated in safe and well-defined ways. For example, when `setTimeout`'s first argument is a string, rather than a function, it exhibits `eval`-like behavior, which violates ADsafety's constraints. Thus we instead give it the type

$$(\mathsf{Widget} \to \mathsf{Widget}) \times \mathsf{Widget} \to \mathsf{Num}$$

Doing so forces the first argument to be a function and, in particular, not a string. Now consider its use:

```
later: function (func, timeout)
/*: Widget × Widget → Widget */ {
  if (typeof func === "function") {
    setTimeout(func, timeout || 0);
  } else { error(); }
}
```

Because `ADSAFE.later` is exported to widgets, it can only assume the Widget type for its arguments, including

$$\{\star\} \text{ is shorthand for } \{\star : F_\star, \mathit{proto} : T_p, \mathit{code} : T_c, f_1 : F_1, \ldots\}$$

$$\begin{array}{lcl} (f_1,\ldots)^+ - (s_1,\ldots)^+ & = & \forall f_i \notin (s_1,\ldots), (f_i,\ldots)^+ \\ (f_1,\ldots)^- - (s_1,\ldots)^+ & = & (f_1,\ldots,s_1,\ldots)^- \end{array} \qquad \begin{array}{ll} f \in (f_1,\ldots)^+ & : \exists f_1 . f = f_1 \\ f \in (f_1,\ldots)^- & : \forall f_1 . f \neq f_1 \end{array}$$

$$\mathit{fields}_\star(\{\star\}, S) = \begin{cases} F_\star & : \begin{array}{c} S_\star \neq \emptyset \text{ and} \\ F_\star \neq \mathsf{Absent} \end{array} \\ \bot & : \text{otherwise} \end{cases} \quad \text{where } S_\star = S - (f_1,\ldots)^+ \qquad \mathit{fields}_p(\{\star\}, S) = \begin{cases} \mathsf{Undef} & : T_p = \mathsf{Null} \\ \mathit{fields}(T_p, S_p) & : S_p \neq \emptyset \\ \bot & : \text{otherwise} \end{cases} \quad \text{where } S_p = S - (f_i \mid F_i \neq \mathsf{Absent})^+$$

$$\begin{array}{lcl} \mathit{fields}(\{\star\}, S) & = & \{T_i \mid f_i \in S \text{ and } F_i = T_i\} \cup \mathit{fields}_\star(\{\star\}, S) \cup \mathit{fields}_p(\{\star\}, S) \\ \mathit{fields}(T_1 \cup T_2, S) & = & \mathit{fields}(T_1, S) \cup \mathit{fields}(T_2, S) \\ \mathit{fields}(T, \emptyset) & = & \bot \end{array}$$

$$\frac{\Sigma;\Gamma \vdash e_o : T_o \qquad \Sigma;\Gamma \vdash e_f : S \qquad S <: \mathsf{Str} \qquad T_{res} = \mathit{fields}(T_o, S)}{\Gamma \vdash e_o[e_f] : T_{res}} \quad \text{(T-LOOKUP)}$$

Figure 8: Typing object lookup

Object Type T_o	String Type S	*fields*(T_o, S)
$\{$*proto* : Null, $\star$: Bool, `"x"` : Num$\}$	$($`"x"`$)^+$	Num
$\{$*proto* : Null, $\star$: Bool, `"x"` : Num$\}$	$($`"x"`, `"y"`$)^+$	Num $\cup$ Bool $\cup$ Undef
$\{$*proto* : Object, $\star$: Num$\}$	$($`"toString"`$)^+$	Num$\cup \rightarrow$ Str
$\{$*proto* : Object, $\star$: Num, `"toString"` : Absent$\}$	$($`"toString"`$)^+$	$\rightarrow$ Str
$\{$*proto* : Null, $\star$: Str, `"x"` : Num, `"y"` : Bool, `"eval"` : ☠$\}$	$($`"eval"`$)^-$	Str $\cup$ Num $\cup$ Bool $\cup$ Undef
$\{$*proto* : Null, $\star$: Str, `"x"` : Num, `"y"` : Bool, `"eval"` : ☠$\}$	$($`"eval"`$)^+$	untypable

Figure 9: Examples of property lookup using *fields*

`func`. A traditional type checker would thus conclude that `func` has type Widget everywhere in `later`. Because Widget includes Str, the invocation of `setTimeout` would yield a type error—even though this is precisely what the conditional in `later` is avoiding!

If-splitting is the name for a collection of techniques that address this problem [39]. Our particular solution uses a refinement of this idea, called flow typing [18], which complements type-checking with flow analysis. The analysis informs the type checker that due to the **`typeof`** check, uses of `func` in the **`then`**-branch of the conditional can in fact be *refined* from the large Widget type of Str $\cup$ Num $\cup \ldots$ to the function type that `setTimeout` requires.

7.2 Required Refactorings

Our type system cannot type check the ADsafe runtime as-is; we need to make 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. To gain confidence that we didn't change ADsafe's behavior, we run ADsafe's sample widgets against our refactored version of ADsafe, and they behave as expected. We describe these refactorings 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.

Inlined `reject_global` Checks Most Bunch methods start by asserting `reject_global(this)`, which ensures that **`this`** is Widget-typed in the rest of the method. Our type system cannot account for such non-local side-effects, but once we inline `reject_global`, if-splitting is able to refine types appropriately (for instance, in the `Bunch.prototype.append` example early in this section).

`makeableTagName` ADsafe's whitelist of safe DOM elements is defined as a dictionary:

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

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:

$$\texttt{document.createElement} : (\texttt{"script"})^{-} \rightarrow \mathsf{HTML}$$

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

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

Our type checker cannot account for this check. We instead refactor the whitelist (a trick noted elsewhere [29]):

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

The type of these strings are $(\texttt{"div"})^{+}$, $(\texttt{"p"})^{+}$,$(\texttt{"b"})^{+}$, etc., so that `makeableTagName[tagName]` has type $(\texttt{"div"}, \texttt{"p"}, \texttt{"b"}, \ldots)^{+}$. Since this finite set of strings excludes `"script"`, it now matches the argument type of `createElement`.

7.3 Cheating 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 an uninteresting type error and wanted to make progress. Our goal, of course, was to ultimately remove every `cheat` from the program.

We were unable to remove two `cheat`s, leaving eleven unverified source lines in the 1,800 LOC ADsafe runtime. We can, in fact, ascribe interesting types to these functions, but checking them is beyond the power of our type system. The details may not be of interest to the general reader, but the web content contains the full body of unverified code and a discussion of its types.

8 ADsafety Redux

Sections 5 and 7 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 1 (section 5.2) 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 9 details one exploit that resulted from such an ad hoc approach), programmers may be well served to use our type checker instead.

Type Soundness Most type systems come with a soundness theorem that is stated as *progress* (well-typed programs do not error) and *preservation* (well-typed programs do not violate their types).

We do not attempt to establish progress. Establishing it would require many more refactorings in the ADsafe runtime, and many lintable widgets would be untypable. Because runtime errors are perfectly acceptable (they halt execution before something bad happens), we relax some of the typing rules in an existing type system [18]—which does exhibit progress—to instead allow some JavaScript errors (e.g., applying non-function values or looking up fields of **null**). We do still need an "untyped progress" theorem that states that our JavaScript semantics fully models all error cases. This theorem is provided by Guha, et al. [17].

We restate and prove preservation for the extensions to Guha et al.'s type system, which is applicable to *all* JavaScript programs.[7] Stated formally:

Lemma 1 (Type Preservation) *If, for an expression e, type T, environment Γ and abstract heap Σ,*

1. $\Sigma \vdash \sigma$,
2. $\Sigma; \Gamma \vdash e : T$, *and*
3. $\sigma e \rightarrow \sigma' e'$;

then there exists a Σ' with $\Sigma' \vdash \sigma'$ and $\Sigma'; \Gamma \vdash e' : T$.

Our assumed environment (section 6) provides the abstract heap Σ and abstract environment Γ, which model the initial state of the browser, σ. Given this lemma, we can make type-based statements about the combination of widgets and `adsafe.js`:

[7]For the formal proof, see Guha et al. [18] and the supplemental material on the web.

Theorem 1 (ADsafety) *For all widgets p, if*

1. *all subexpressions of p are* Widget*-typable,*
2. `adsafe.js` *is typable,*
3. `adsafe.js` *runs before p, and*
4. *$\sigma p \to \sigma' p'$ (single-step reduction),*

then at every step p', p' also has the type Widget*.*

This theorem says that for all widgets p whose subexpressions are Widget-typed, if `adsafe.js` type-checks and runs in the browser environment, p can take any number of steps and still have the Widget type. Since types are preserved, two further key lemmas hold during execution:

Lemma 2 (Widgets cannot load new code at runtime) *For all widgets e, if all variables and sub-expressions of e are* Widget*-typed, then e does not load new code.*

By section 6, `eval`-like functions are ☠-typed, 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. ■

Lemma 3 (Widgets do not obtain DOM references) *For all widgets e, if all variables and sub-expressions of e are* Widget*-typed, then e does not obtain direct* DOM *references.*

The type of DOM objects is not subsumed by the Widget type. All functions in the ADsafe runtime have the type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \to \mathsf{Widget}$$

Thus, functions in the ADsafe runtime do not leak DOM references, as long as they are only applied to Widget-typed values. Since all subexpressions of the widget e are Widget-typed, all values that e passes to the ADsafe runtime are Widget-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.

```
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 10: Exploiting JSLint

Widgets cannot communicate This claim is false; section 9 presents a counterexample.

9 Bugs Found in ADsafe

We have implemented the type system presented in this paper, and applied it to the ADsafe source. The implementation is about 3,000 LOC, and takes 20 seconds to 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 7.2. The other failures, however, represent genuine errors in ADsafe that were present in the production system. The same applies to instances where JSLint and our typed model of it failed to conform. 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:

```
fakeNode = {
  appendChild: function(elt) {
    myWindow = elt.ownerDocument.defaultView;
  }
};
```

The full exploit is in figure 10.

```
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", o);
});
```

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

Figure 11: Firefox-specific Exploit for ADsafe

This bug manifested as a discrepancy between our model of JSLint as a type checker and the real JSLint. Recall from section 5 that all expressions in widgets must have type Widget (defined in figure 4). For `{ "__nodes__": [fakeNode] }` to type as Widget, the `"__nodes__"` field must have type Array⟨HTML⟩∪Undef. However, `[fakeNode]` has type Widget, 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. (An alternative solution would be to use our type system as a replacement for JSLint.) We note that when the ADsafe option of JSLint was first announced,[8] its author offered:

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

After this error report, he confirmed, "I do believe that I owe you a plate of shrimp".

Missing Runtime Checks Many functions in `adsafe.js` incorrectly assumed that they were applied 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 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[9] that contains arbitrary JavaScript (figure 11).

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 Widget × Widget → Widget, and thus the type of `value` is Widget.

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.

Counterexamples to Non-Interference Finally, a type error in `Bunch.prototype.getStyle` helped us generate a counterexample to ADsafe's claim of widget non-interference (definition 1, part 4). The `getStyle` method is available to widgets, so its type must be Widget → Widget. The following code is the essence of `getStyle`:

```
Bunch.prototype.getStyle = function (name) {
  var sty;
  reject_global(this);
  sty = window.getComputedStyle(this.__node__);
  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 a reference to the prototype of the browser's `CSSStyleDeclaration` objects. Thus the return type of the body is not Widget, yielding a type error.

A widget cannot exploit this bug in isolation. However, it can replace built-in methods of CSS style objects

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

[9] https://developer.mozilla.org/en/XBL

and interfere with the operation of the hosting page and other widgets that manipulate styles in JavaScript.

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.

Prior Exploits Before and during our implementation, other exploits were found in ADsafe and reported [27–29]. We have run our type checker on the exploitable code, and our tools catch the bugs and report type errors.

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, `adsafe.js` also underwent non-security related refactorings during the course of this work. Despite not providing its author our type checker, we were easily able to continue 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 Widget type and checked. In short, our type checker has shown robustness in the face of program edits.

10 Beyond ADsafe

Our security type system is capable of verifying useful properties about JavaScript programs in general. Sections 5, 6, and 7 present carefully crafted *types* that we ascribe to the browser API and `adsafe.js`, and use to model widget programs. Proving these types hold over the ADsafe runtime library and JSLint-ed widgets guarantees robust sandboxing properties for ADsafe.

Verifications for other sandboxes would require the design of new *types*, to accurately model checked, rewritten programs and their interface to the sandbox, but not necessarily a new *type system*. Indeed, our type-based strategy provides a 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 by type-checking.

In particular, developers of *new* sandboxes should be aware of this strategy. 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 8).

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

Reasoning About Strings Our type system lets programmers reason about finite sets of strings and use these sets to lookup fields in objects. To verify Caja, we would need to reason about string patterns. For example, Caja uses the field named `"foo"+ "_w__"` to store a flag that determines if the field `"foo"` is writable.

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 [39], 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.

11 Related Work

Verifying JavaScript Web Sandboxes ADsafe [9], BrowserShield [35], Caja [33], and FBJS [13] 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 [17, 26].

Maffeis et al. [27] 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 [30]. 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. [38] develop a flow analysis to find bugs in the ADsafe runtime (that we also catch). They simplify the analysis by modeling ECMAScript 5 strict mode, which is not fully implemented in any current Web browser. In contrast, ADsafe is designed to run on current browsers, and thus supports older and more permissive versions of JavaScript. We use the semantics and tools of Guha, et al. [17], which does not limit itself to the strict mode, so 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 [31, 34] 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 paper and the aforementioned works are founded on detailed JavaScript semantics.

Yu, et al. [40] use JavaScript sandboxing techniques to enforce various security policies on untrusted code. Their semantic model, CoreScript, simplifies the 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 paper.

Modeling the Web Browser There are formal models of Web browsers that are tailored to model whole-browser security properties [1, 6]. These do not model JavaScript's semantics in any detail and are therefore orthogonal to semantic models of JavaScript [17, 26] 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 Analysis of JavaScript GateKeeper [15] 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. [8] and VEX [5] 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 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 proven sound.

Other static analyses for JavaScript [16, 21, 22] are not specifically designed to encode and check security.

Type Systems Our type checker is based on that of Guha, et al. [18]. Theirs has a restrictive type system for objects that we fully replace to type check ADsafe. We also add simple extensions to their *flow typing* system to account for additional kinds of runtime checks employed by ADsafe. Their paper surveys other JavaScript type systems [2, 19] that can type-check other patterns but have not been used to verify security-critical code, which is the goal of this paper. Our treatment of objects is also derived from ML-ART [36], but accounts for JavaScript features and patterns such as function objects, prototypes, and objects as dictionaries.

Language-Based Security Schneider et al. [37] survey the design and type-based verification of language-based security systems. JavaScript Web sandboxes are inlined reference monitors [12]. Guha, et al. [17] offer a type-based strategy to verify these, but their approach—which depends on building a custom type rule around each check in the reference monitor—does not scale to a program of the size of ADsafe. Furthermore, their custom rules essentially hand-code if-splitting, which we obtain directly from the underlying type system.

Cappos, et al. [7] 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.

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 [20]. 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 [10, 11, 23–25, 32, 40]. 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.

Acknowledgments

We thank Douglas Crockford for discussions, openmindedness, and insightful feedback (and the promise of certain crustaceans); Mark S. Miller for enlightening discussions; Matthias Felleisen, Andrew Ferguson, and David Wagner for numerous helpful comments that helped us understand weaknesses in exposition; the NSF for financial support; and StackOverflow, as well as Claudiu Saftoiu (our lower-latency version of StackOverflow), for unflagging attention to detail.

References

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *IEEE Computer Security Foundations Symposium*, 2010.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[3] J. 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.

[4] I. Awad, T. Close, A. Felt, C. Jackson, B. Laurie, F. Lee, K.-P. Lee, D.-S. Hopwood, J. Nagra, E. Sachs, M. Samuel, M. Stay, and D. Wagner. Caja external security review. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/Caja_External_Security_Review_v2.pdf`.

[5] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. 2010.

[6] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Usenix Conference on Web Application Development (WebApps)*, 2010.

[7] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. 2009.

[9] D. Crockford. ADSafe. `www.adsafe.org`, 2011.

[10] A. Dewald, T. Holz, and F. C. Freiling. ADSandbox: Sanboxing JavaScript to fight Malicious Websites. In *Symposium On Applied Computing (SAC)*, 2010.

[11] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference*, 2009.

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

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

[14] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Network and Distributed System Security Symposium*, 2010.

[15] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium (SSYM)*, 2009.

[16] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *International World Wide Web Conference*, 2009.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, 2011.

[19] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2009.

[20] C. Jackson and H. J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *International World Wide Web Conference*, 2007.

[21] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, 2009.

[22] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, 2010.

[23] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. In *International World Wide Web Conference*, 2007.

[24] E. Kıcıman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. 2007.

[25] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In *USENIX Security Symposium (SSYM)*, 2010.

[26] S. Maffeis, J. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *ASIAN Symposium on Programming Languages and Systems*, pages 307–325, 2008.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[28] S. Maffeis, J. C. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *W2SP'09*. IEEE, 2009.

[29] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*. IEEE, 2010.

[30] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. 2010.

[31] J. Magazinius, P. H. Phung, and D. Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *OWASP AppSec Research*, 2010.

[32] L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[33] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. 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`.

[34] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. 2009.

[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Symposium on Operating Systems Design and Implementation*, 2006.

[36] D. Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Springer Lecture Notes in Computer Science*, pages 321–346. Springer Berlin / Heidelberg, 1994.

[37] F. B. Schneider, G. Morrisett, and R. Harper. A Language-Based Approach to Security. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Springer Lecture Notes in Computer Science*, pages 86–101. Springer Berlin / Heidelberg, 2001.

[38] A. Taly, Ú. Erlingsson, M. S. Miller, J. C. Mitchell, and J. Nagra. Automated analysis of security-critical JavaScript APIs. 2011.

[39] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 395–406, 2008.

[40] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[41] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In *International World Wide Web Conference*, 2009.

A Typing and JSLint: An Example

Consider the following malicious widget:

```
var fakeBunch = {
  __nodes__ : [ {
    appendChild: function(elt) {
      myWindow = elt.ownerDocument.parentView;
    } ],
  tagName : ``p''
}
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 12: The Unverified Portion of ADsafe

It is malicious because the ADsafe runtime could be tricked into calling `appendChild` with a direct DOM reference. Fortunately, this is rejected by both JSLint and by a Widget checker, but for very different reasons.

JSLint rejects it because `__nodes__` is a banned field. In contrast, the Widget-checker assumes that the variable `elt` is Widget-typed. This is sufficient to type-check the body of the function, whose type (Undef) is subsumed by Widget. Hence, the function has type Widget $\rightarrow$ Widget, which is subsumed by Widget. Similarly, the `__nodes__` field types to Widget and the object literal has type:

```
{
  __nodes__ : Widget,
  tagName : Widget
}
```

However, the object literal, since it is a subexpression, must be Widget-typed too. Widget requires the object's `__nodes__` field to have type Array⟨HTML⟩, which does not match the Widget type calculated above. Hence, we have a type error and the widget is rejected.

Relative to standard type machinery, JSLint is much more ad hoc. For instance, it rejects harmless programs whose variables happen to be banned names, because it has no contextual information. Thus, there are safe widgets that the Widget-checker accepts that JSLint rejects. Worse, the ad hoc nature of JSLint results in errors, such as the missing static check in section 9.

B Unverifiable Code

Figure 12 presents the entire unverified codebase (reformatted for column width), which we now discuss.

`reject_name` The `reject_name` function returns **`true`** if its argument is a blacklisted field and **`false`** otherwise. With a little work, we can give it a type like String $\rightarrow$ Bool. However, `reject_name` is used in predicate positions to guard against invalid field accesses; therefore, it needs to return a more precise type like

$$\mathsf{UnsafeField} \rightarrow \mathsf{True} \cap \mathsf{SafeField} \rightarrow \mathsf{Bool}$$

where UnsafeField and SafeField are, respectively, string-set types of blacklisted names and their complement. The above type lets us conclude that if the predicate returns false, the argument was *not* a blacklisted field, and can thus be dereferenced safely. The if-splitter (section 7) uses this information.

Unfortunately, checking that the body of `reject_name` has this type is too sophisticated for our if-splitter. To type it, we would need a solver that incorporates the semantics of `charAt` and other primitives. Since we lack that, we use a `cheat` to ascribe this type, and verify it by manual inspection.

`ADSAFE.create` In the (new) 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 current browsers through `ADSAFE.create`. This function is never used by ADsafe; it is only intended for widgets. Therefore, its type must be

$$[\mathsf{Global} \cup \mathsf{Widget}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

JSLint ensures that the actual argument is Widget-typed (section 5). However, the return type is problematic. In our Widget type (figure 4), the $proto$ field admits Object but not Widget, which is necessary to type-check the code. Permitting α (which represents Widget) in the type of $proto$ results in a type system that we have not been able to show will terminate.

`ADSAFE._intercept` ADsafe enables the hosting Web page to provide *interceptors*, which are functions that get direct access to the DOM. The above count excludes interceptors because, by definition, these are unsafe. Verifying interceptors requires analyzing the whole of the *page*, including its HTML and how it modifies ADsafe, which are outside the scope of our work.