ADsafety

Type-Based Verification of JavaScript Sandboxing

Joe Politz[*]

Abstract

Web sites routinely incorporate programs from several sources into a single page. These sources must be protected from one another, which requires robust sandboxing of JavaScript. 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, in which we found several bugs and other weaknesses.

1 Introduction

A *mashup* is a Web page that 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, yet mashups are significantly more pervasive. Indeed, Web pages that display advertisements from ad networks are also mashups, since they often employ JavaScript for animations and interactivity. A survey of popular pages shows that a large percentage include scripts from a diverse array of external sources [41]. Unfortunately, these third-party code fragments 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 ads.

There are various ways to secure mashups; this paper addresses language-based Web sandboxing systems, most of which have similar high-level goals and designs (section 2). In section 3, we review the design and implementation of sandboxes, and witness the need for tool-supported verification. Section 4 provides a detailed plan for the rest of the paper.

[0]Submitted in partial completion of degree requirements of the Brown Computer Science PhD program, February 2011. Joint work with Spirodon Eliopolous, Arjun Guha, and Shriram Krishnamurthi.

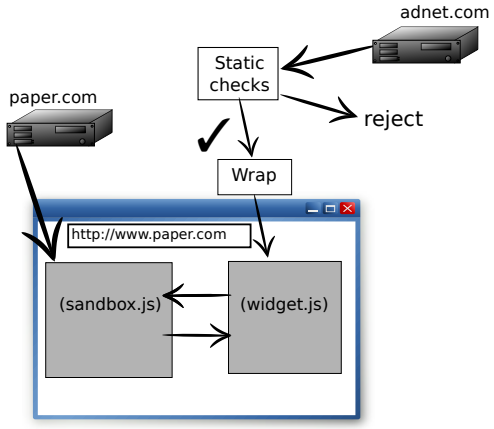

Figure 1: Web sandboxing architecture

2 Language-based Web Sandboxing

The Web browser environment provides references to objects that implement network access, disk storage, geolocation, and other powerful capabilities. These are used legitimately by rich Web applications, but they can be exploited by embedded widgets since all JavaScript on a page runs in the same global environment. A Web sandbox thus restricts or attenuates access to these capabilities, allowing trusted containers to safely embed untrusted widgets. ADsafe [9], Caja [33], FBJS [13], and BrowserShield [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 as definitely 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, it is rewritten to include calls to the runtime library. The runtime library, and the widget (which has been checked and rewritten), must both 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, to trust a Web sandbox, we must precisely state its notion of security and prove 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 walk through an imaginary review of a Web sandbox, focusing on the details of ADsafe. Later, we will discuss how to (mostly) remove the humans from the loop.

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

- A static verifier, called JSLint[1], 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.

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

Attenuated Capabilities There are various capabilities in the browser environment that widgets should not be able to directly reference. Direct DOM references are particularly dangerous; 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 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 container designates a DOM subtree for the widget 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 let the widget obtain wrappers to elements outside its subtree.
- The runtime wraps many DOM functions that are conditionally unsafe. For example, `document.createElement` is usually safe, unless it 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*[2]. 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

[2] A Bunch wraps a collection of HTML elements.

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]`

need to be verified. However, Bunches 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 work around 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 that the static checker admits 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 circumvent dynamic safety checks if they 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, 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 by preventing widgets from accessing fields that are introduced by the sandbox. For example, ADsafe stores direct DOM references in the `__nodes__` field of Bunches. The `__nodes__` field is also blacklisted.

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 ADsafe: Our Roadmap

Because humans are expensive and error-prone, and the code-review needs to be repeated every time the program changes, it is best to automate the review process. 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 an innovative type checker [18] intended for traditional type-checking of JavaScript.[3]

Before we begin verification, we need some definition of what security means for ADsafe. From correspondence with its 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.*

The assumption on built-in prototypes is often violated in practice [14]. Nevertheless, like ADsafe, we make this assumption; mitigating it is outside our scope.

Our goal is to use type-based arguments to prove ADsafety. We mostly achieve this (section 8) after various bugs exposed by our type checker (section 9) are fixed. 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 these.

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

5 Modeling JSLint

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 only on the security-critical static JavaScript checks in JSLint.

[3] See `http://cs.brown.edu/~joe/temp/a9s8jfew/research-comps/joe-research-comps.html` for our implementation and other details.

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

Figure 3: Type Language for ADsafe and Widgets

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

Figure 4: The Widget type

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.[4] 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

[4] 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.

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, which are trivially typed:

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

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

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

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:

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

The full list of blacklisted fields is in figure 4. Our type checker signals a type error on any ☠-typed field access. The Ref indicates that the object is mutable. We use a recursive type (μ) to indicate that the other fields, $\star$, may recursively contain Widget-typed values.[5] JSLint tries to ensure that objects in widgets do not have `toString` and `valueOf` properties. Our Absent type has the same effect, ensuring that these fields are not covered by $\star$.

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:

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

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

The type of the *code* field indicates that widget-functions may have an arbitrary number of Widget-typed arguments and return Widget-typed results.[6] It also specifies that the type of the implicit **`this`**-argument (written inside brackets) may be either Widget or Global. This latter type models the underlying reason for JSLint's rejection of all widgets that contain **`this`**. 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 lookup in JavaScript implicitly accesses the prototypes, we specify the type of prototypes in Widget:

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

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 that are used internally by `adsafe.js` to build the Bunch abstraction. To keep our types simple, we enumerate these three fields instead of pattern-matching on field names:

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

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

[6] 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.

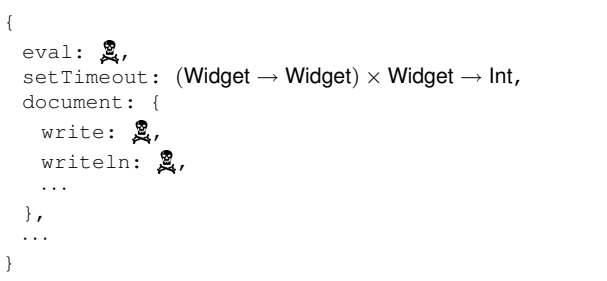

Figure 5: A Fragment of the Type of `window`

ideally 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 enabled) accepts a widget e, then e can be typed with all its variables and sub-expressions* 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. Therefore, our type checker could, in principle, be used as a replacement for JSLint for ADsafe's purposes. Our goal, however, is to ensure that Widget corresponds to what JSLint admits in practice, and the above tests give us this confidence. Appendix A discusses how type-checking with Widget differs internally from the operation of JSLint.

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. We now discuss how we model the language and the browser.

A Tractable JavaScript Semantics We use the semantics of Guha, et al. [17], which reduces JavaScript to a core semantics called λ_{JS}, which models the "essentials" of JavaScript—prototype-based objects, first-class functions, basic control operators, and mutation. λ_{JS} thus omits many of JavaScript complexities.

However, λ_{JS} comes with a *desugaring* function that maps all JavaScript programs (idiosyncrasies included) to equivalent λ_{JS} programs that explicitly encode 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.

Naturally, we should wonder whether desugaring correctly accounts for the idiosyncrasies of JavaScript. Guha, et al. answer this by testing 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 claims:

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 ensure it is not, we mark `eval` with from section 5, which marks banned fields. However, 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 never 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. However, 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 ADsafe Runtime

The Widget type, developed in section 5, 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 from `adsafe.js`:

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

```
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

We might assume that the `child` argument 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 [Bunch]Bunch → Bunch. Instead, the most precise type we can ascribe is:

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

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, like 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 → 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, since these types are not compatible with Widget.

T-StringSet
$$\Sigma;\Gamma \vdash str : (str)^+$$

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

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

ST-StringUnion
$$\frac{\forall f \in (f_1,\ldots,f_1',\ldots), f \in (s_1,\ldots)}{(f_1,\ldots)^+ \cup (f_1',\ldots)^+ <: (s_1,\ldots)^+}$$

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

Figure 7: Typing and operations on string set types

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 have 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.

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 which 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. Thus we instead give it the type

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

This 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 `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 ∪ Num ∪ ... to the function type that `setTimeout` requires.

7.2 Required Refactorings

Our type system is not expressive enough to prove the safety of 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. 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 `append` example above).

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

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

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

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

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

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

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

$$\mathit{fields}_\star(\{\star\}, S) = \begin{cases} F_\star & : S_\star \neq \emptyset \text{ and } F_\star \neq \mathsf{Absent} \\ \bot & : \text{otherwise} \end{cases} \quad \text{where } S_\star = S - (f_1, \cdots)^+$$

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

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

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

Figure 8: Typing object lookup

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

Figure 9: Examples of property lookup using *fields*

$(\texttt{"div"}, \texttt{"p"}, \texttt{"b"}, \cdots)^+$. 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 only 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 appendix B 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. However, 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.

Lemma 1 (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 2 (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:

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

Thus, functions in the ADsafe runtime does not leak DOM references, as long as they 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.

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

Type Preservation 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). Here we present a statement of the theorem of type preservation for combining widgets and `adsafe.js`:

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. $p \rightarrow p'$ *(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 thus preserved, the above lemmas of ADsafety hold true at all times during widget execution.

What About Progress? End-user type systems reject programs that signal runtime errors. However, our security type system does not, because they are irrelevant to ADsafety. In fact, runtime errors are perfectly acceptable—they halt execution before anything bad can happen. Therefore, our type system explicitly allows programs to signal JavaScript errors (e.g., applying non-function values or looking up fields of **null**). We could catch such errors also, but doing so would require many more refactorings in the ADsafe runtime, and many lintable widgets would be untypable.

However, we 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].

9 Bugs Found in ADsafe

We have implemented a type checker for 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) {
```

```
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

```
    myWindow = elt.ownerDocument.defaultView;
  },
};
```

The full exploit is in figure 10.

This bug manifested itself as a discrepancy between our model of JSLint as a type checker and the real JSLint. Recall that we model JSLint as a type checker (section 5) by requiring all expressions in widgets to have the type Widget (defined in figure 4). For `{ "__nodes__": [fakeNode] }` to type to 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,[7] 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[8] 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.

```
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

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

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

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 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 offending exploitable examples, 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

The programming patterns in ADsafe are shared with other Web sandboxes (section 3). We thus believe that our type checker could be extended to verify them, too.

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. ADsafe is designed to run on current browsers, which support older and more permissive versions of JavaScript. This paper uses the semantics and tools of Guha, et al. [17] that do not make such simplifications. We thus find new bugs in the ADsafe runtime. In addition, Taly, et al. use a very simplified model of JSLint. In contrast, we provide a detailed, type-theoretic account of JSLint, and also test it. We 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. However, the soundness of its runtime checks is left for future work. In contrast, we verify ADsafe's runtime checks using a type system. Our type system also models ADsafe's static verifier, which uses lightweight syntactic checks, which are much simpler than the semantic properties that program analyses are designed to model.

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.

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 very 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.

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. In *USENIX Conference on Security*, 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. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

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

[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. `wiki.developers.facebook.com/index.php/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. 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. In *Symposium on Operating System Principles*, 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. Runtime 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. In *IEEE Symposium on Security and Privacy*, 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. In *ACM Symposium on Information, Computer and Communications Security*, 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. In *IEEE Symposium on Security and Privacy*, 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*, 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"
}
```

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 typecheck 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:

```
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

```
{
  __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 → 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

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

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 7.1) 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

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

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.