Run-Time Enforcement of Secure JavaScript Subsets

Sergio Maffeis
Imperial College London
maffeis@doc.ic.ac.uk

John C. Mitchell
Dep. of Computer Science
Stanford University
mitchell@cs.stanford.edu

Ankur Taly
Stanford University
ataly@stanford.edu

Abstract

Many Web-based applications such as advertisement, social networking and online shopping benefit from the interaction of trusted and unstrusted content within the same page. If the untrusted content includes JavaScript code, it must be prevented from maliciously altering pages, stealing sensitive information, or causing other harm. We study filtering and rewriting techniques to control untrusted JavaScript code, using Facebook FBJS as a motivating example. We explain the core problems, provide JavaScript code that enforces provable isolation properties at run-time, and compare our results with the techniques used in FBJS.

1 Introduction

Many contemporary web sites incorporate untrusted content. For example, many sites serve third-party advertisements, allow users to post comments that are then served to others, or allow users to add their own applications to the site. Although untrusted content can be placed in an isolating `iframe` [3], this is not always done because of limitations imposed on communication between trusted and untrusted code. Instead, Facebook [19], for example, preprocesses untrusted content, applying filters and source-to-source rewriting before the content is served. While some of these methods make intuitive sense, JavaScript [7, 9] provides many subtle ways for malicious code to subvert language-based isolation methods, as shown here and in our previous work [15].

In this paper, we review some previous filtering methods for managing untrusted JavaScript [15] and explore ways of replacing some aspects of these restrictive static code filters with more flexible run-time instrumentation that is implementable as source-to-source translation. Our previous efforts uncovered problems and vulnerabilities with the then-current versions of FBJS and ADsafe [5], Yahoo's safe advertising proposal. We then developed a formal foundation for proving isolation properties of JavaScript programs [15], based on our operational semantics of the full ECMA-262 Standard language (3rd Edition) [6], available on the web [12] and described previously in [14]. The language subsets defined in [15] provided a foundation for code filtering – any JavaScript filter that only allows programs in a meaningful sublanguage will guarantee any semantic properties associated with it. More specifically, we developed proofs that certain subsets of the ECMA-262 Standard language make it possible to syntactically identify the object properties that may be accessed, make it possible to safely rename variables used in the code, and/or make it possible to prevent access to scope objects (including the global object). However, these syntactic subsets are more restrictive than the solution currently employed by Facebook, which uses run-time instrumentation to restrict the semantic behavior of code that would not pass our filters. In this paper, we therefore focus on subsets of JavaScript and semantic restrictions that model the effect of rewriting JavaScript source code with "wrapper" functions. Our main contribution is the definition of JavaScript code that implements secure, semantic preserving run-time checks that enforce isolation of untrusted JavaScript code. We also compare our methods with the solutions employed by Facebook a the time of our submission. In particular, we describe a previously unknown Facebook vulnerability that we discovered thanks to our analysis, and the fix adopted in the current version of FBJS following our disclosure to them.

Related work on language-based methods for isolating the effects of potentially malicious web content include [17], which examines ways to inspect and cleanse dynamic HTML content, and [25], which modifies questionable JavaScript, for a more restricted fragment of JavaScript than we consider here. A short workshop paper [24] also gives an architecture for server-side code analysis and instrumentation, without exploring details or specific methods for constraining JavaScript. The Google Caja [4] project follows instead a different approach, based on transparent compilation of JavaScript code into a capability-based JavaScript subset, with libraries that emulate DOM objects.

Additional related work on rewriting based methods for controlling the execution of JavaScript include [11]. Foundational studies of limited subsets of JavaScript and dynamic languages in general are reported in [2, 22, 25, 10, 18, 1, 23]; see [14].

2 JavaScript Isolation Problems

In this Section, we summarize the Facebook isolation mechanism. Over time, several teams of researchers have discovered flaws in the Facebook protection mechanisms that were promptly addressed by the Facebook team [8, 16, 15]. Specific handling of $FBJS.ref described below, for example, is the result of vulnerabilities reported to Facebook [15]. Based on past evidence, we believe it is important to develop a foundation for proving isolation properties. Without careful scrutiny and reliable semantic methods, it is simply not possible to reliably reason about a programming language as complex as JavaScript.

2.1 Facebook JavaScript

Facebook is a web-based social networking application. Registered and authenticated users store private and public information on the Facebook website in their Facebook profile, which may include personal data, list of friends (other Facebook users), photos, and other information. Users can share information by sending messages, directly writing on a public portion of a user profile (called the wall), or interacting with Facebook applications.

Facebook applications can be written by any user and can be deployed in various ways: as desktop applications, as external web pages displayed inside a frame within a Facebook page, or as integrated components of a user profile. Integrated applications are by far the most common, as they affect the way a user profile is displayed.

Facebook applications are written in FBML [21], a variant of HTML designed to make it easy to write applications and also to restrict their possible behavior. A Facebook application is retrieved from the application publisher's server and embedded as a subtree of the Facebook page document. Since Facebook applications are intended to interact with the rest of the user's profile, they are not isolated inside an `iframe`. However, the actions of a Facebook application must be restricted so that it cannot maliciously manipulate the rest of the Facebook display, access sensitive information (including the browser cookie) or take unauthorized actions on behalf of the user. As part of the Facebook isolation mechanism, the scripts used by applications must be written in a subset of JavaScript called FBJS [20] that restricts them from accessing arbitrary parts of the DOM tree of the larger Facebook page. The source application code is checked to make sure it contains valid FBJS, and some rewriting is applied to limit the application's behavior before it is rendered in the user's browser.

FBJS. While FBJS has the same syntax as JavaScript, a preprocessor consistently adds an application-specific prefix to all top-level identifiers in the code, isolating the effective namespace of an application from the namespace of other applicantions and of the rest of the Facebook page. For example, a statement document.domain may be rewritten to a12345_document.domain, where a12345_ is the application-specific prefix. Since this renaming will prevent application code from directly accessing most of the host and native JavaScript objects, such as the document object, Facebook provides libraries that are accessible within the application namespace. For example, the libraries include the object a12345_document, which mediates interaction between the application code and the true document object.

Additional steps are used to restrict the use of the special identifier this in FBJS code. The expression this, executed in the global scope, evaluates to the window object, which is the global scope itself. Without further restrictions, an application could simply use an expression such as this.document to break the namespace isolation and access the document object. Since renaming this would drastically change the meaning of JavaScript code, occurrences of this are replaced with the expression $FBJS.ref(this), which calls the function $FBJS.ref to check what object this refers to when it is used. If this refers to window, then $FBJS.ref(this) returns null.

Other, indirect ways that malicious content might reach the window object involve accessing certain standard or browser-specific predefined object properties such as __parent__ and constructor. Therefore, FBJS blacklists such properties and rewrites any explicit access to them in the code into an access to the useless property __unknown__. Since the notation o[e] denotes the access to the property of object o whose name is the result of evaluating expression e to a string, FBJS rewrites that term to a12345_o[$FBJS.idx(e)], where $FBJS.idx enforces blacklisting on the string value of e. Note that this technique is not vulnerable to standard obfuscation, because $FBJS.idx is run on the string obtained as the final result of evaluating e.

Finally, FBJS code runs in an environment where properties such as valueOf, which may access (indirectly) the window object, are redefined to something harmless.

2.2 Formalizing JavaScript Isolation

FBJS illustrates two fundamental issues with mashup isolation. (i) Regardless of the technique adopted to enforce isolation, the ultimate goal is usually very simple: make sure that a piece of untrusted code does not access a certain set of global variables (typically the DOM). (ii) While enforcing this constraint may seem easy, there are a number

of subtleties related to the expressiveness and complexity of JavaScript.

Common isolation techniques include blacklisting certain properties, separating the namespaces corresponding to code in different trust domains, inserting run-time checks to prevent illegal accesses, and wrapping sensitive objects to limit their accessibility.

In the remainder of this paper, we study how combining run-time checks (analogous to $FBJS.idx and $FBJS.ref) with syntactic restrictions leads to expressive and provably secure subsets of JavaScript. While we use FBJS as a running example, the ideas illustrated in this paper also apply to JavaScript isolation in other settings.

3 Syntactic JavaScript Subsets

In this Section, we describe two secure subsets of JavaScript (first defined in [15]) that enforce isolation exclusively by means of syntactic restrictions, so that the user code is directly executed in the browser. The informal properties stated in this section are all fully supported by formal proofs available in [15]. These earlier results are included in the present paper both as background for modifications to them we present in Section 4, and as motivation for more permissive, run-time checks in the user code.

Two JavaScript Isolation Problems. If we can solve the problem of determining the set of properties that a piece of code can access, then we can isolate global variables by a simple syntactic check.

Our first subset, Jt, is designed to solve this problem without restricting the use of this. A JavaScript program can get hold of its own scope by way of this. For example, the expression var x; this.x=42 effectively assigns 42 to variable x. In fact, manipulating the scope leads to a confusion of the boundary between variables (which are properties of scope objects) and properties of regular object. Hence, Jt code must be prevented from using as property name any of the global variable names to be protected. In theory, this does not constitute a significant limitation of expressiveness. Effectively, Jt is a good subset for isolating the code of a single untrusted application from a library of functions whose names may be all prefixed by a designated string such as $. On the other hand, Jt is not suited to run several applications with separate namespaces, since the sets of property names used by each one needs to be disjoint.

To better support multiple applications, the next problem we have to solve is to prevent code from explicitly manipulating the scope, so that variables are effectively separated from regular object properties. To this end, we propose a refinement of Jt, which we call Js, that forbids the use of this. Hence, only the global variable names of each application, and of the page libraries, need to be distinct from one another. Moreover, Js enjoys the property that the semantics of its terms does not change after a safe renaming of variables. Hence, isolation can be enforced by an automatic rewriting pass (with suitable side-conditions).

3.1 Isolating property names: Jt

The problem of determining the set of properties names that may be accessed by a piece of code is intractable for JavaScript in general, because property names can be computed using string operations, as in o={prop:42}; m="pr"; n="op"; o[m + n], which returns 42. However, we can determine a finite set containing all accessed properties if we eliminate operations that can convert strings to property names, such as eval and e[e]. In doing so, we must also consider implicit access to native properties that may not be mentioned explicitly in the code. For example, the code fragment var o = { }; "an "+ o causes an implicit type conversion of object o to a string, by an implicit call to the toString property of object o, evaluating to the string "an [object Object]". (If o does not have the toString property, then it is inherited from its prototype). Fortunately, the property names that can be accessed implicitly are only the natural numbers used to index arrays and a finite set of native property names [14].

Definition 1 *The set $\mathcal{P}_{nat}$ of all the property names that can be accesses implicitly is* {0,1,2,...} $\cup$

$$\left\{ \begin{array}{l} \textit{toString, toNumber, valueOf, constructor, prototype,} \\ \textit{length, arguments, message, Object, Array, RegExp} \end{array} \right\}$$

This list is exhaustive for an ECMA-262-compliant implementation. Other properties may be added to $\mathcal{P}_{nat}$ to account for browser-specific JavaScript extensions.

Our first subset, called Jt, is designed to make property access (whether for read or for write) decidable.

Definition 2 *Jt is defined as JavaScript minus all terms containing the identifiers eval, Function, hasOwnProperty, propertyIsEnumerable and constructor; the expressions e[e], e in e; the statement for (e in e) s.*

Since we consider checking for the existence of a property as a read access, we exclude from Jt also the e in e and for (e in e) s statements, even though they cannot be used to read the actual contents of the corresponding property.

From the usability point of view, the only serious restrictions of Jt are the lack of eval, and e[e]. The former, although has practical uses, is commonly considered *evil*, and is excluded from most subsets. The latter constitutes the natural way to access arrays elements. The dynamic subset Jb of Section 4.1 addresses this limitation.

Jt lends itself naturally to enforce *whitelisting* of properties and variable. It can also be used to enforce *blacklisting*. A Jt piece of code cannot read or write any variable or

property, except for those in $\mathcal{P}_{nat}$, that does not appear explicitly in its code or in a function pre-loaded in the run-time environment (Theorem 1 of [15]). A simple static analysis can be used to screen the actual code for blacklisted properties. Since the initial JavaScript environment is defined by the specification, blacklisting can be effectively enforced as long as the code of any pre-loaded, user-defined function is known *a priori* (such is the case for Facebook).

3.2 Protecting the Scope: *Js*

In ECMA-262-compliant JavaScript implementations there are three ways to obtain a pointer to a scope object. The simplest way, supported by all JavaScript implementations, is by referencing the global object, for example by evaluating the expression this in the global scope. Another way to get a pointer to a scope object is by the statement

```
try {throw (function(){return this})}
catch(get_scope){scope=get_scope(); ...};
```

When the code is executed, the function thrown as an exception in the try block is bound to the identifier get_scope in a new scope object that becomes the scope for the catch block. Hence, when we call get_scope(), the this identifier of the function is bound to the enclosing scope object, which we make available to arbitrary code by saving it in variable scope. Although this behaviour conforms to the ECMA-262 standard, as far as we are aware Safari, Opera and Chrome are the only browser where this example works. Other browsers, such as for example Internet Explorer and Firefox bind the global object instead of the catch scope object to the this of the call to get_scope in the catch clause. Finally, we can get a pointer to a scope object by the expression

```
(function get_scope(x){if (x==0) {return this}
                       else {scope = get_scope(0); ...}})(1)
```

Here we use a named function expression. As this function executes, the static scope of the recursive function is a fresh scope object where the identifier get_scope is bound to the function itself, making recursion possible. When in the else branch we recursively call get_scope(0), then this is once again bound to the scope object, which is saved in scope for later usage. Once again, although ECMA-262-compliant, this example works only in Firefox and Safari. Internet Explorer, Opera and Chrome instead bind the global object to the this of get_scope in the recursive call.

We now define the subset *Js* which keeps variables distinct from property names by preventing manipulation of explicit scope objects (Theorem 2 of [15]).

Definition 3 *The subset Js is defined as Jt minus all terms containing this, with(e){s} and the identifiers valueOf, sort, concat and reverse.*

First and foremost the subset forbids any use of this, which can be used to access scope objects as described above. Just like in FBJS, we need to remove also the with construct because it gives another (direct) way to manipulate the scope. For example, the code var o = {x:null}; with(o){x=42} assigns 42 to the property o.x. Since we eliminate this and with, scope objects are only accessible via internal JavaScript properties which in turn can only be accessed as a side effect of the execution of other instructions. For example, the internal scope pointer of a scope object is accessed during identifier resolution, in order to search along the scope chain. However, its value is never returned as the result of evaluating a term. Similarly, the scope pointer stored in a function closure is never returned as a result. The internal @this property is returned only by the reduction rule for this, which cannot be triggered in *Js*, and by the native functions concat, sort or reverse of Array.prototype, and valueOf of Object.prototype. For example, the expression valueOf() evaluates to window (which is also the initial scope). By defining *Js* as a subset of *Jt*, we can blacklist these dangerous properties.

Closure under renaming The goal of variable renaming is to isolate the namespaces of different applications without requiring all of the property names to be distinct. Therefore, we want o.p to be renamed to a12345_o.p, and not to a12345_o.a12345_p. Due to implicity property access, and the fact that variables are effectively undistinguishable from properties of scope objects, the definition of variable renaming in JavaScript is subtle. In particular, one should not rename all the variables that correspond to native properties of a scope object, including the ones inherited via the prototype chain. These properties in fact have a predefined semantics that cannot be preserved by renaming. For example toString() evaluates to *"[object Window]"*, but throws a "reference error" exception when evaluated as a12345_toString() after renaming.

Since *Js* does not contain with, only the global object, internal activation objects or freshly allocated objects (in the case of try-catch and named functions) can play the role of scope objects. Hence, the only (non-internal) inherited native properties are the ones present in Object.prototype, and the pre-defined properties of the global object. The complete set of properties that should not be renamed, denoted by $\mathcal{P}_{noRen}$ is:

{ NaN,Infinity,undefined,eval,parseInt,parseFloat,IsNaN,
IsFinite,Object,Function,Array,String,Number,Boolean,
Date,RegExp,Error,RangeError,ReferenceError, TypeError,
SyntaxError,EvalError,constructor,toString,toLocaleString,
valueOf,hasOwnProperty,propertyIsEnumerable,isPrototypeOf }

Bowser implementations contain additional properties such as document,setTimeout,etc..

Let a *safe renaming* be a partial injective function that renames identifiers (not in $\mathcal{P}_{noRen}$) without introducing clashes. In [15], we prove that the intended meaning of a Js program does not change under renaming. Jt instead does not support the semantics preserving renaming of variables. The counterexample try {throw (function(){return this});} catch(y){y().x=42; x;} is valid Jt code that, according to the JavaScript semantics, evaluates to 42. If we rename x to $x, in the catch clause is rewritten to catch(y){y().x=42; $x} which raises an exception because $x is undefined.

3.3 Comparison with FBJS

A purely syntactic solution to the FBJS isolation problem, justified by our analysis, is to restrict Facebook applications to Js. While this could be an attractive solution for isolating user-supplied applications in contexts where code is written from scratch, it is more restrictive than the solutions proposed in Section 4. Since Js preserves safe renamings, we can separate the namespaces of different applications, and of the FBJS libraries, without altering their semantics. Since it is a subset of Jt, a simple syntactic check on application code guarantees that it cannot escape its namespace or access blacklisted properties (which need to include also browser-specific extensions such as caller, __proto__, getters, setters, etc.).

FBJS is more expressive than Js, because it includes a (sanitized) version of this and of the member access e[e] notation. On the other hand, FBJS does not correctly support renaming because it does not prevent explicit manipulation of the scope, and because it renames the properties in $\mathcal{P}_{noRen}$. The toString and try-catch counterexamples of Section 3.2 apply to FBJS as well. In Section 4 we shall propose better subsets that preserve renaming and are as expressive as FBJS.

4 Semantic JavaScript Subsets

In this Section, we present three JavaScript subsets that, by virtue of using run-time checks, are more expressive than Jt and Js yet still enforce strong insolation properties. The informal claims put forward in this Section are proven in the online version [13].

JavaScript Isolation Problems Revisited. While the subset Jt of Section 3 makes it possible to statically determine all the properties accessed during execution of given code, this subset prevents e1[e2], which is often useful in programming. We therefore define a subset Jb with modified semantics (wrapper function) that allows e1[e2] and guarantees the weaker property that no program accesses properties that are explicitly blacklisted.

Our second semantic subset, called Js^s, is the semantic counterpart to Js. It solves the same problem of preventing the direct manipulation of scope objects, but it is more expressive, because Js^s programs can use this when it does not evaluate to a scope object. Disallowing this altogether would break many existing JavaScript libraries, and entail extensive rewriting.

The last semantic subset of this section, called Jg (first defined in [15]), solves the problem of isolating the window object, hence the global scope, while permitting to use this, even when it is bound to other scope objects. Indeed, we shall see that for some purposes the ability to explicitly manipulate the scope can be a desirable.

4.1 Blacklisting Properties: Jb

We now define the subset Jb that prevents user code from accessing any property included in a blacklist (or excluded from a whitelist). Note that if a property in $\mathcal{P}_{nat}$ is blacklisted it can still be accessed implicitly as a side effect.

Definition 4 *Let $\mathcal{B}$ be a set of blacklisted properties. The subset $Jb(\mathcal{B})$ is defined as Jt plus the construct e[e], minus all terms containing property names or identifiers in $\mathcal{B}$.*

In order for $Jb(\mathcal{B})$ to effectively achieve its isolation goal, $\mathcal{B}$ must contain at least the properties eval, Function and constructor blacklisted also by Jt, and a small number of private identifiers beginning with $, as explained below.

Enforcing Jb. The idea is to insert a run-time check in each occurrence of e1[e2] to make sure that e2 does not evaluate to a blacklisted property name. We transform every access to a blacklisted property of an object into an access to the property *"bad"* of the same object (we assume that $\mathcal{B}$ does not contain *"bad"*). A different option, clashing with the JavaScript *silent failure* philosophy is to throw an exception when a blacklisted property is accessed.

A faithful implementation of Jb is complicated by subtle details of the JavaScript semantics for the expression e1[e2]. In fact, the execution of e1[e2] goes through several steps involving evaluation of expressions to values, and possibly type conversions executed in a very specific order. Roughly, first e1 is evaluated to a value va1, then e2 to va2, then if va1 is not an object it is converted into an object o, and similarly if va2 is not a string it is converted into a string m:

$$\texttt{e1[e2]} \longrightarrow \texttt{va1[e2]} \longrightarrow \texttt{va1[va2]} \longrightarrow \texttt{o[va2]} \longrightarrow \texttt{o[m]}$$

Each of these steps, which precede the actual access of property m in o, may raise an exception or have other side effects. Therefore, their execution order must be preserved.

The simplest and most efficient faithful implementation of this run-time check that we could find is to rewrite e1[e2] to e1[IDX(e2)], where IDX(e2) is the expression

($=e2,{toString:function(){return($=TOSTRING($),FILTER($))}})

The IDX code evaluates once and for all e2 to a value va2 that is saved in the variable $, and returns an object value va so that effectively the internal execution steps so far are

e1[IDX(e2)] ⟶ va1[IDX(e2)] ⟶ va1[va] ⟶ o[va]

Since va is an object and not a string, its toString method is invoked next. The expression TOSTRING($), which is defined as (new $String($)).valueOf() converts va2 into a string. In fact, the most direct way to convert a value into a string exactly as o[va2] would do, is by passing va2 to the original String constructor (which we assume to have saved in a variable $String), and invoking the valueOf method of the resulting string object. Finally, the expression FILTER($), defined as

```
($ == "$String" ? "bad" :
          ($ == "$" ? "bad" :
                    ($ == "constructor" ? "bad" : $))
```

uses nested conditional expressions to return the string saved in $ if it is not in the blacklist $\mathcal{B}$, and "bad" otherwise. For this filtering to work $, $String and constructor must always be blacklisted (and cannot appear as identifiers or property names in the source code). While these are the only blacklisted properties in the code above, it is straightforward to nest further conditional expressions to blacklist other properties. An alternative implementation of FILTER($) is the expression ($blacklist[$]?"bad":$), where $blacklist is a (blacklisted) global variable containing an object with the properties to be blacklisted initialized to true. Note that all the properties of Object.prototype that are not overridden by $blacklist, and that do not contain values (such as null,0,"",false) that evaluate to false in a boolean context, will be automatically blacklisted. Hence, in our case $blacklist should actually be the object

```
{$:true,$String:true,$blacklist:true,
 toString:false,toLocaleString:false,...}
```

Our run time check is correct with either choice of FILTER.

Claim 1 *For every blacklist $\mathcal{B}$ containing the property names $ and $String, and for every JavaScript program $P \in Jb(\mathcal{B})$, the program $String=String;Q where Q is obtained by rewriting every instance of e1[e2] in P to e1[IDX(e2)] (adapted to include all of $\mathcal{B}$), behaves exactly like P when P accesses non-blacklisted properties. If P accesses a blacklisted property m of an object o, Q accesses instead o["bad"].*

In many practical cases, one can use simpler variants of IDX, sacrificing their correspondence to the original semantics of e1[e2].

When the order of the side-effects (including exceptions) caused by the evaluation of e1 and e2 can be ignored (say because the exceptions are not caught, or the expressions are side-effect free) we can simplify IDX(e2) to be ($=TOSTRING(e2),FILTER($)).

If e2 evaluates to an object va2, converting va2 to a string in the expression o[va2] involves invoking first its toString method, and if that fails, its valueOf method. The opposite happens when converting va2 to a string by the expression va2+"". If va2.toString() returns the same value as va2.valueOf(), or if the latter does not return a string, we can redefine TOSTRING(e2) in IDX to be the expression e2+"".

Combining these two simplifications, we can define IDX(e2) as ($=e2+"",($blacklist[$]?"bad":$)). which is remarkably simple and efficient, and in particular implements correctly the JavaScript semantics in the most common case when the expression e2 is just a string or a number.

These latest variants of IDX do not enjoy Claim 1 because there are some (corner) cases in which their behaviour departs from that of e1[e2]. Yet, they are secure, because they still prevent any blacklisted property from being accessed.

4.2 Protecting the Scope: Js^s

In Js, we exclude this because it can be used to obtain a scope object. Now, we reinstate this and look for dynamic ways to prevent it to be bound to scope objects.

Definition 5 *The subset Js^s is defined as Jt minus all terms containing with(e){s}, the identifiers valueOf, sort, concat and reverse and property names or identifiers beginning with $.*

Js^s still excludes valueOf, sort, concat and reverse because those native functions can return the window object, if called in the appropriate context.

Enforcing Js^s. Unfortunately, it is not possible to enforce Js^s in an ECMA-262-compliant implementation of JavaScript. In the general case, there is no JavaScript expression that can detect if an object has an internal scope pointer, or test for its existence directly. Only code that has a handle to a scope object *that is present in the scope chain* can test such object and detect that it is a scope object. Recall the two ways of obtaining a scope object described in Section 3.2. In the case of the recursive function, the scope object that we obtain is active in the scope chain just below the activation object of the function returning its this. Therefore, we can insert a run-time check that detects it and replaces it with null. In the try-catch case instead the function returning its this is defined before the scope of the catch branch is created, so when at run-time the catch scope object is bound to the this, it is not active in the (static) scope chain of the function, and cannot be detected.

Hence, our implementation is useful to prevent direct scope manipulation in Firefox, which as discussed in Section 3.2 returns a scope only in the recursive function case, but not in Safari or other strictly ECMA-262-compliant implementations, which return the scope also in the try-catch.

To enforce Js^s in Firefox all we need to do is to initialize a global (blacklisted) variable $ with true, and replace each

instance of this with the expression NOSCOPE(this), defined as (this.$=false,$?(delete this.$,this):(delete this.$,$=true,null)). When this is bound to the global object, the expression this.$=false overrides the global declaration, which needs to be restored by the $=true expression in the last branch of the conditional. In the case of a local scope object, this expression leaves behind a useless (but unharmful) local binding of $ to true. In the case of regular objects, the temporary variable $ is correctly removed.

Claim 2 *For every Firefox-JavaScript program $P \in Js^s$ that does not contain $, the program $=true;Q where Q is obtained by rewriting every instance of this in P to NOSCOPE(this), behaves exactly like P when P never accesses a this bound to a scope object. If P evaluates the expression this to a scope object then Q evaluates the same expression to null.*

4.3 Isolating the Global Object: Jg

In Section 4.2 we argued that, in general, it is not possible to detect a scope object in an ECMA-262-compliant JavaScript implementation. What we can do instead, is to prevent this to be bound to the global object. This solution is effectively equivalent to Js^s for Internet Explorer, because in that browser local scope objects cannot be accessed anyway, as discussed in Section 3.2. In the other browsers, keeping at least the global variables separate from generic property names still supports flexible isolation policies, as discussed for Js.

Arguably, the ability to manipulate scope objects directly may be a desirable feature. For example, it can be used to implement *open closures* which are a concept that we discovered after understanding direct scope manipulation via the examples given in Section 3.2. The idea is to write expressions that return a number of functions sharing some private state (like normal closures), *plus* an object that effectively embodies that shared state. A software architecture may distribute such functions, guaranteeing the encapsulation of the shared state, plus retain a handle to the shared state itself. In particular, in the case where the functions participating in the closure return results by updating shared variables, the shared state is ready to be used as a result object, without need to do any copying. For example, given

```
var oc = (function scope(x){if (x==0) {return this}
  else {shared=scope(0);shared.y=7;
  return [function(){y+=23},function(){y+=12},shared]}})(1)
```

the expression oc[0]();oc[1]();oc[2].y evaluates to 42. Traditional closures could encode less efficiently some of this behaviour by providing a dedicated function to access and update the shared state.

The subset Jg contains this and isolates the global object.

Definition 6 *The subset Jg is defined as Js plus the this expression and minus all terms containing property names or identifiers beginning with $.*

Note that Jg still excludes valueOf, sort, concat and reverse, that can return the window object.

Since the local scope can still be directly manipulated, in general variables can be confused with property names, and therefore variable renaming does not preserve the meaning of programs. Yet, this rarely happens accidentally, and does not constitute a security problem. On the other hand, since variables defined in the global scope *are* effectively separated from property names, Jg can be used to isolate the namespaces of different applications.

Enforcing Jg. In practice, the semantic restriction can be implemented by rewriting every occurrence of this in the user code into the expression NOGLOBAL(this) defined as (this==$?null;this). $ is a blacklisted global variable, initialized with the address of the global object.

Claim 3 *For every JavaScript program $P \in Jg$ that does not contain $, the program $=this;Q where Q is obtained by rewriting every instance of this in P to NOGLOBAL(this), behaves exactly like P if P does not access a this bound to the global object. If P evaluates this to the global object then Q evaluates NOGLOBAL(this) to null.*

4.4 Comparison with FBJS

We now compare our run-time checks with the corresponding ones in FBJS. Below, we denote by $FBJS^v_{09}$ the version of FBJS deployed on Facebook at the time of our analysis, in March 2009. The $FBJS^v_{09}$ $FBJS.ref function carries out a check equivalent to NOGLOBAL, plus some additional filtering needed to wrap DOM objects exposed to user code (we reserve to study the secure wrapping of libraries in future work). Since $FBJS is effectively blacklisted in $FBJS^v_{09}$, we are satisfied that ref prevents the this identifier to be evaluated to the window object, and the check is semantically faithful in the spirit of Claim 3.

The $FBJS^v_{09}$ $FBJS.idx function instead does not preserve the semantics of the member access notation, and as a result can be compromised. In the context of our explanation of Section 4.1, $FBJS.idx is in fact equivalent to the expression ($=e2,($instanceof Object||$blacklist[$])?"bad":$), where $blacklist is the object {caller:true,$:true,$blacklist:true}. The main problem is that, differently from our definition of IDX, the expression $blacklist[$]?"bad":$ converts va (that in principle could be an object) to a string two times. The object

```
{toString:function(){this.toString=function(){return "caller"};
                     return "good"}}
```

can fool the blacklisting by first returning the good property *"good"*, and then returning the bad property *"caller"* (we found a similar attack, which has since been fixed, on [11]). To avoid this problem, $FBJS^{v}_{09}$ inserts the check $ instanceof Object that tries to detect if $ contains an object. In general, this check is not sound. According to the JavaScript semantics, any object with a null prototype (such as Object.prototype) escapes this check. Moreover, in Firefox, Internet Explorer and Opera also the window object escapes the check.

In $FBJS^{v}_{09}$, Object.prototype and window are not accessible by user code, so cannot be used to implement this attack. We found instead that the scope objects described in Section 3.2 have a null prototype in Safari, and therefore we were able to mount attacks on the $FBJS.idx that effectively let user application code escape the Facebook sandbox. (See [15] for examples of exploit code, and a discussion on the security implications.) Shortly after our notification of this problem, Facebook has modified the $FBJS.ref function to include code that detects if the current browser is Safari, and in that case checks if this is bound to an object able to escape the instanceof check described above.

Unfortunately this solution is not very robust, and is unnecessarily restrictive. First, some browsers may have other host objects that have a null prototype, and that can be accesses without using this. Such objects could still be used to subvert $FBJS.idx, which has not been changed. Second, $FBJS.idx prevents objects to be used as arguments of member expressions. This restriction is unnecessary for the safety of blacklisting, as shown by our IDX.

Another minor problem with $FBJS.idx is that it deals inconsistently with the blacklisting of inherited properties such as toString. While the expression ({}).toString() is valid FBJS code returning *"[object␣Object]"*, the expression ({})[*"toString"*]() raises an exception because toString is implicitly blacklisted.This problem can be easily fixed, as described in Section 4.1, by setting $blacklist.toString=false.

5 Conclusions

We reviewed previous filtering methods for managing untrusted JavaScript and developed ways of replacing restrictive static code filters with more flexible run-time instrumentation that is implementable as source-to-source translation. We defined a subset with modified semantics (wrapper functions) that allows e1[e2] and guarantees that no program accesses properties that are explicitly blacklisted. Our second semantic subset prevents the direct manipulation of scope objects, but allows programs to use this when it does not evaluate to a scope object. Our third semantic subset isolates the window object, and hence the global scope, while permitting code to use this, even when it is bound to other scope objects. We have applied our results to analyze FBJS, which apart from some minor problems discovered by our analysis, has proven to be a remarkably sound and efficient practical JavaScript subset. We hope that our semantics-based study will convince developers of the value of programming language methods for evaluating language-based isolation.

Acknowledgments. Sergio Maffeis is supported by EPSRC grant EP/E044956 /1. Mitchell and Taly acknowledge the support of the National Science Foundation.

References

[1] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proc. of FM 2008*, volume 5014 of *LNCS*, pages 262–277. Springer, 2008.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of ECOOP'05*, pages 429–452, 2005.

[3] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *Proc. of USENIX Security*, 2008.

[4] Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based web. http://code.google.com/p/google-caja/.

[5] Douglas Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org/, 2008.

[6] ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, 1999.

[7] B. Eich. JavaScript at ten years. http://www.mozilla.org/js/language/ICFP-Keynote.ppt.

[8] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proc. of SocialNets '08*. ACM, 2008.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006.

[10] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. Proc. of FOOL'09, 2009.

[11] P. H.Phung, D. Sands, and A. Chudnov. Lightweight self protecting JavaScript. In *Proc. of ASIACCS 2009*. ACM Press, 2009.

[12] S. Maffeis, J.C. Mitchell, and A. Taly. Complete ECMA 262-3 operational semantics. http://jssec.net/semantics/.

[13] S. Maffeis, J.C. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. Long version with proofs available online. April, 2009.

[14] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[15] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[16] J. Pynnonen. Facebook script injection vulnerabilities. http://seclists.org/fulldisclosure/2008/Jul/0023.html.

[17] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.

[18] A. Sabelfeld and A. Askarov. Tight enforcement of flexible information-release policies for dynamic languages. Proc. of PCC'08, 2008.

[19] The FaceBook Team. FaceBook. http://www.facebook.com/.

[20] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[21] The FaceBook Team. FBML. http://wiki.developers.facebook.com/index.php/FBML.

[22] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, pages 408–422, 2005.

[23] P. Thiemann. A type safe DOM API. In *Proc. of DBPL'05*, pages 169–183, 2005.

[24] K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Proc. of W2SP'08*, 2008.

[25] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. of POPL'07*, pages 237–249, 2007.