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

Web sites that incorporate untrusted content may use browser- or language-based methods to keep such content from maliciously altering pages, stealing sensitive information, or causing other harm. We use accepted methods from the study of programming languages to investigate language-based methods for filtering and rewriting 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 [7], 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 [6, 9] provides many subtle ways for malicious code to subvert language-based isolation methods, as shown here and in our previous work [**?**].

In this paper, we review some previous filtering methods for managing untrusted JavaScript [**?**] 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 [**?**], based on our operational semantics of the full ECMA-262 Standard language (3rd Edition) [11], available on the web [12] and described previously in [13, 14]. The language subsets defined in [**?**] 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.

Related work on language-based methods for isolating the effects of potentially malicious web content include [18], 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 safe subset with libraries that emulate DOM objects.

Additional related work on rewriting based methods for controlling the execution of JavaScript include [16]. Foundational studies of limited subsets of JavaScript and dynamic languages in general are reported in [2, 22, 25, 10, 19, 1, 23]; see [13].

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, 17, 15]. Specific handling of $FBJS.ref described below, for example, is the result of vulnerabilities reported to Facebook [**?**]. 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 [**?**]) 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 [**?**]. 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* $\{0,1,2,...\} \bigcup$

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

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 fully supports *blacklisting* of properties and variables. 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 [**?**]). 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 is by evaluating the expression this in the global scope, which yields the global object. This is a very direct access that is supported by all JavaScript implementations. 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 [**?**]).

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:

$$\left\{\begin{array}{l}\text{NaN,Infinity,undefined,eval,parseInt,parseFloat,IsNaN,}\\ \text{IsFinite,Object,Function,Array,String,Number,Boolean,}\\ \text{Date,RegExp,Error,RangeError,ReferenceError, TypeError,}\\ \text{SyntaxError,EvalError,constructor,toString,toLocaleString,}\\ \text{valueOf,hasOwnProperty,propertyIsEnumerable,isPrototypeOf}\end{array}\right\}$$

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 [**?**] 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 Appendix A.

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 [?]), 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. 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 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:

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

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

$$\text{e1[IDX(e2)]} \longrightarrow \text{va1[IDX(e2)]} \longrightarrow \text{va1[va]} \longrightarrow \text{o[va]}$$

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 : all terms containing this; 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". To avoid this problem, $FBJS^v_{09}$ inserts the check $ instanceof Object that tries to detect if $ contains an object. In general, this check 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. Shortly after our notification of this problem, the $FBJS.ref function has been modified to include a check that detects the current browser, and if it is Safari it then checks if this is bound to an object able to escape the instanceof check described above.

Unfortunately this piecemeal solution is not very satisfactory, for two reasons. 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 hope that this semantics-based study of JavaScript isolation 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 *FM 2008*, volume 5014 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2008.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of ECOOP'05*, page 429452, 2005.

[3] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *17th USENIX Security Symposium*, 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] B. Eich. JavaScript at ten years. www.mozilla.org/js/language/ICFP-Keynote.ppt.

[7] FaceBook. Web Site. http://www.facebook.com/.

[8] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *SocialNets '08*, pages 25–30, New York, NY, USA, 2008. ACM.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006. http://proquest.safaribooksonline.com/0596101996.

[10] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. Foundations of Object-Oriented Languages (FOOL'09), 2009.

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

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

[13] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325. Springer Verlag, December 2008.

[14] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[15] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. Submitted for publication, February 2009.

[16] David Sands Phu H.Phung and Andrey Chudnov. Lightweight self protecting JavaScript. In *ASIACCS 2009*. ACM Press, 2009.

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

[18] 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.

[19] A. Sabelfeld and Aslan Askarov. Tight enforcement of flexible information-release policies for dynamic languages. Second International Workshop on Proof-Carrying Code'08, 2008.

[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*, page 408422, 2005.

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

[24] K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Web 2.0 Security & Privacy (W2SP)*, 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.

A Appendix : Correctness Proofs

In this Section we formally prove the correctness of our enforcement mechanisms. In order to prove the correctness, we make use of the operational semantics of JavaScript. The main purpose of this appendix is to substantiate the claims made in Section 4. The reader does not need to read this Appendix in order to understand the main body of the paper. Unless otherwise stated, below we assume that the semantics of JavaScript is compliant with the ECMA-262 standard.

A.1 Operational Semantics of JavaScript

We briefly summarize our formalization of the operational semantics of JavaScript [12, 13, 14] based on the ECMA-262 standard [11], and introduce some auxiliary notation and definitions. In [13], we proved properties of JavaScript that address the internal consistency of the semantics itself, and memory reachability properties needed for garbage collection, but did not address the kind isolation properties.Discussion of the relation between this semantics and current browsers implementations appear in [13]

Our operational semantics consists of a set of rules written in a conventional meta-notation suitable for rigorous but (currently) unautomated proofs. Given the space constraints, we describe only the main semantic functions and some representative axioms and rules.

Semantic Functions and Contexts. Expressions, statements and programs each have a corresponding small-step semantic relation denoted respectively by $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$. Each semantic function transforms a heap H, a pointer in the heap to the current scope l, and the current term being evaluated t into a new heap-scope-term triple.

The semantics of programs depends on the semantics of statements which in turn depends on the semantics of expressions which in turn, for example by evaluating a function, depends circularly on the semantics of programs. These dependencies are made explicit by contextual rules, that specify how a transition derived for a term can be used to derive a transition for a larger term including the former as a sub-term. The premises of each semantic rule are predicates that must hold in order for the rule to be applied, usually built of very simple mathematical conditions such set membership, inequality and semantic function application.

An atomic transition is described by an axiom. For example, the axiom H,l,(v) $\longrightarrow$ H,l,v describes that brackets can be removed when they surround a value (as opposed to an expression, where brackets are still meaningful). Contextual rules propagate such atomic transitions. For example, if program H,l,P evaluates to H1,l1,P1 then also H,l,@FunExe(l2,P) (an internal expression used to evaluate the body of a function) evaluates to H1,l1,@FunExe(l2,P1).

The rule below shows that: @FunExe(l,−) is one of the contexts eCp for evaluating programs.

$$\frac{\text{H,l,P} \xrightarrow{P} \text{H1,l1,P1}}{\text{H,l,eCp[P]} \xrightarrow{e} \text{H1,l1,eCp[P1]}}$$

Expressions. We distinguish two classes of expressions: internal expressions, which correspond to specification artifacts needed to model the intended behavior of user expressions, and user expressions, which are part of the user syntax of JavaScript. Internal expressions include addresses, references, exceptions and functions such as @GetValue,@PutValue used to get or set object properties, and @Call,@Construct used to call functions or to construct new objects using constructor functions.

Statements. Similarly to the case for expressions, the semantics of statements contains a certain number of internal statements, used to represent unobservable execution steps, and user statements that are part of the user syntax of JavaScript. A completion is the final result of evaluating a statement.

co ::= "("ct,vae,xe")"   vae::=&empty|va   xe::=&empty|x
ct ::= Normal | Break | Continue | Return | Throw

The completion type indicates whether the execution flow should continue normally, or be disrupted. The value of a completion is relevant when the completion type is Return (denoting the value to be returned), Throw (denoting the exception thrown), or Normal (propagating the value to be return during the execution of a function body). The identifier of a completion is relevant when the completion type is either Break or Continue, denoting the program point where the execution flow should be diverted to.

Programs. Programs are sequences of statements and function declarations.

P ::= fd [P] | s [P]   fd ::= function x "("[x˜]")"{"[P]"}"

As usual, the execution of statements is taken care of by a contextual rule. If a statement evaluates to a break or continue outside of a control construct, an SyntaxError exception is thrown (rule (i)). The run-time semantics of a function declaration instead is equivalent to a no-op (rule (ii)). Function (and variable) declarations should in fact be parsed once and for all, before starting to execute the program text. In the case of the main body of a JavaScript program, the parsing is triggered by rule (iii) which adds to the initial heap NativeEnv first the variable and then the function declarations (functions VD,FD).

$$\frac{\begin{array}{l}\text{ct} < \{\text{Break,Continue}\}\\ \text{o = new\_SyntaxError()} \quad \text{H1,l1 = alloc(H,o)}\end{array}}{\text{H,l,(ct,vae,xe) [P]} \xrightarrow{P} \text{H1,l,(Throw,l1,\&empty)}} \quad \text{(i)}$$

$$\text{H,l,function x ([x˜])\{[P]\} [P1]} \xrightarrow{P} \text{H,l,(Normal,\&empty,\&empty) [P1]} \quad \text{(ii)}$$

$$\frac{\begin{array}{l}\text{VD(NativeEnv,\#Global,\{DontDelete\},P) = H1}\\ \text{FD(H1,\#Global,\{DontDelete\},P) = H2}\end{array}}{\text{P} \xrightarrow{P} \text{H2,\#Global,P}} \quad \text{(iii)}$$

Native Objects. NativeEnv is the initial heap of core JavaScript. It contains native objects for representing predefined functions, constructors and prototypes, and the global object @Global that constitutes the initial scope, and is always the root of the scope chain. In web browsers, the global object is called window. For example, the global object defines properties to store special values such as &NaN and &undefined, functions such as eval and constructors to build generic objects, functions, numbers, booleans and arrays. Since it is the root of the scope chain, its @Scope property points to null. Its @this property points to itself. None of the non-internal properties are read-only or enumerable, and most of them can be deleted.

A.2 Preliminaries

We now define some notation and state some properties of the semantics that support the formal analysis of JavaScript subsets defined in Section 4.

A *state* S is a triple (H, l, t). We use the notation $\mathcal{H}(S)$, $\mathcal{S}(S)$ and $\mathcal{T}(S)$ to denote each component of the state. We denote by H_0 the "empty" heap, that contains only the native objects, and no user code. We use l_G to denote the heap address of the global object #Global. If a heap, a scope pointer and a term are well-formed then the corresponding state is also well-formed (see the Appendix of [?] for a formal definition). In [14], we show that the evaluation of well-formed terms, if it terminates, yields either a value or an exception (for expressions), or a completion (for statements and programs). A state S is *initial* if it is well-formed, $\mathcal{H}(S) = H_0$, $\mathcal{S}(S) = l_G$ and $\mathcal{T}(S)$ is a user term. A *reduciton trace* τ is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S_1 \to \ldots \to S_n \to \ldots$. Give a state S, we denote by $\tau(S)$ the (unique) trace originating from S and, if $\tau(S)$ is finite, we denote by $Final(S)$ the final state of $\tau(S)$.

To ease our analysis, we add a separate sort mp to distinguish property names from strings and identifiers in the semantics. We make all the implicit conversions between these sorts explicit, by adding the identity functions Id2Prop: x ⟶ mp, Prop2Id: mp ⟶ x; Str2Prop: m ⟶ mp, Prop2Str: mp ⟶ m. The semantics already contained explicit conversion of strings to programs: ParseProg, ParseFunction, ParseParams. In order to keep track of the names appearing in a state S, we define functions that collect respectively the identifiers and the property names of the term and the heap of S.

$$\mathcal{N}_I^T(S) = \{x | x \in \mathcal{T}(S)\} \quad \mathcal{N}_P^T(S) = \{mp \mid mp \in \mathcal{T}(S)\}$$
$$\mathcal{N}_I^H(S) = \{x \mid x \in P,\ P \in \mathcal{H}(S)\}$$
$$\mathcal{N}_P^H(S) = \{mp \mid \exists l : mp \in \mathcal{H}(S)(l)\}$$
$$\mathcal{N}_I(S) = \mathcal{N}_I^T(S) \cup \mathcal{N}_I^H(S) \quad \mathcal{N}_P(S) = \mathcal{N}_P^T(S) \cup \mathcal{N}_P^H(S)$$

Finally, we define the set of all the identifiers and property names appearing in a state S by $\mathcal{N}(S) = \mathcal{N}_I(S) \cup \text{Prop2Id}(\mathcal{N}_P(S))$. From these definitions, follows that for any initial state S_0, $\mathcal{N}(S_0) = \mathcal{N}_I^T(S_0) \cup \mathcal{N}_P^H(S)$. $\mathcal{N}_P^H(S)$ is the set of property names present in the initial heap H_0. This is a fixed set, and will henceforth be denoted by $\mathcal{N}_P^0$.

We define *meta-call* a pair $(f, (args))$ where f is a semantic function or predicate appearing in the premise of a reduction rule, and $(args)$ is the list of its actual arguments as instantiated by a reduction step using that rule. For every state S, we denote by $\mathcal{C}_1(S)$ the set of the meta-calls triggered directly by a one step transition from state S. Since each meta-call may in turn trigger other meta-calls during its evaluation, we denote by $\mathcal{C}(S)$ the set of all the meta-calls involved in a reduction step. We denote by $\mathcal{F}_H$ the set of functions that can read or write to the heap: $\mathcal{F}_H = \{$Dot(H, l, mp), Get(H, l, mp), Update(H, l,mp), Scope(H, l, mp), Prototype(H, l, mp)$\}$,

Definition 7 *(Property access) For any state S, we define the set of all property names accessed during a single transition by $\mathcal{A}(S) \triangleq \{mp \mid \exists f \in \mathcal{F}_H\ \exists H, l : (f, (H, l, mp)) \in \mathcal{C}(S)\}$. In the case of a trace τ, $\mathcal{A}(\tau) \triangleq \bigcup_{S_i \in \tau} \mathcal{A}(S_i)$.*

In Sections 3 and 4, we considered syntactic subsets of JavaScript. A syntactic subset J is essentially a subset of JavaScript user terms. For a given subset J, we denote by $Initial(J)$, the set of all well-formed initial states for J. We denote by J^* the set

$$J^* = \{t' \mid t \in J \ \wedge\ \exists H, l : H_0, l_G, t \rightarrow H, l, t'\}$$

of all terms that are reachable by reducing terms in J. We denote by $Wf_J(S)$ the well-formedness predicate for a state in the subset J, defined exactly like $Wf(S)$ except that $Wf_{\mathcal{T}}(\mathcal{T}(S))$ instead of checking if a term is derivable by the grammar, checks if the term is in J^*.

A.3 Proof of Claim 1

In this Subsection we prove Claim 1, which states that given a black list $\mathcal{B}$, for all initial states S_0 in the set $Jb(\mathcal{B})$, for which the $\mathcal{T}(S_0)$ is appropriately rewritten, no property from the blacklist $\mathcal{B}$ is accessed. We start by giving a few notations and definitions that will be used in the lemmas and theorems that come later. We define $Jb^r(\mathcal{B})$ as the subset $Jb(\mathcal{B})$ where for every term $t \in Jb(\mathcal{B})$, the subexpression e1[e2] (if it is present in t) is replaced with e1[IDX(e2)]. We formalize the property that if the execution of a program P involves accessing property mp of some object then either $mp \in \mathcal{P}_{nat}$ or $mp \notin \mathcal{B}$ as follows:

Definition 8 *(Pb) Given a well-formed state $S \in Jb^r(\mathcal{B})$, $Pb(S)$ holds iff $\mathcal{A}(\tau(S)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$*

Theorem 1 *For all well-formed states S_0 in $Initial(Jb^r(\mathcal{B}))$, $Pb(S_0)$ holds.*

It is easy to see that theorem 1 proves Claim 1. In this rest of this Subsection we sketch out the proof of Theorem 1. We split the proof into the following two main steps

Step 1: We define a state predicate $Pb^{strong}(S)$ and show that for all initial states S_0 in JavaScript, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$

Step 2: For all initial states S_0 in the subset $Jb^r(\mathcal{B})$, $Pb^{strong}(S_0)$ holds.

A.3.1 Step (1)

Given a blacklist $\mathcal{B}$ we define the following whiteness predicate on states:

Definition 9 *(State Whiteness) For a well-formed state S in the subset $Jb^r(\mathcal{B})^*$, $White(S)$ is true iff $(\mathcal{N}_P^T(S_1) \cup$ Id2Prop$(\mathcal{N}_I(S_1))) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$.*

Consider any reduction rule from the operational semantics. The general structure of such a rule is $\frac{(Premise)}{S_1 \rightarrow S_2}$. We define the following goodness property on rules.

Definition 10 *(Rule goodness) A reduction rule of the form $\frac{(Premise)}{S_1 \rightarrow S_2}$ is good iff for all applicable S_1, S_2*

$$White(S_1) \Rightarrow White(S_2)$$

Based on the above definition of rule goodness we try to enumerate the set of good rules. If the initial state is white and the final state is not white then it is necessarily the case that some additional property names or identifier names get dynamically generated during the particular reduction step. According to our semantics, for most of the reduction rules, all the property names and identifier names that appear in the final state are a subset of those that appear in the initial state. If any identifier present in the final state is not present in the initial state then it must have been obtained by conversion from a string value present in the initial state. Similarly if any property name that appears in the final state, does not appear as a property name or an identifier in the initial state and also does not appear in the set $\mathcal{P}_{nat}$, then it must have been obtained by conversion from a string value present in the initial state. Therefore we claim that if a rule is good then it must not involve any of the following conversions: (1) strings to property names: rule E−ctx−Str−Pname (2) strings to identifiers: rule N−Funparse−StrId; (3) strings to programs: rules N−Funparse−StrProg, E−Eval−StrProg.

The rule E−ctx−Str−Pname also includes the context l[−]. We argue that the context l[−] is not bad if the result of converting the string to a property name is outside the blacklist. In order to make the analysis simpler, we split the rule for the term l∗m in two:

$$\frac{\begin{array}{l}\text{mp = convStrPname(m)}\\ \text{mp !< Blacklist OR mp = "bad"}\end{array}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}} \quad [\text{E}-\text{AccGood}]$$

$$\frac{\text{mp = convStrPname(m) AND mp < Blacklist}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}} \quad [\text{E}-\text{AccBad}]$$

Therefore the rule E−AccGood is good as it applies to only those cases where the resulting property names are outside the blacklist. We define the set $\mathcal{R}^{good}$ as all rules minus the set { E−AccBad, E−ctx−Str−Pname, N−Funparse−StrId, N−Funparse−StrProg, E−Eval−StrProg}. The detailed description for these rules is given in Figure 1.

Lemma 1 *All reduction rules present in the set $\mathcal{R}^{good}$ are good.*

Proof. We divide the set of good rules into two sets: $\mathcal{R}^{good} \setminus \{E - AccGood\}$ and $\{E - AccGood\}$.
Case 1: $\mathcal{R}^{good} \setminus \{E - AccGood\}$
For all rules in this set, we make use of Lemma 1 from [?] which states that all rules of the form $\frac{(Premise)}{S_1 \to S_2}$ in the set $\mathcal{R}^{good} \setminus \{E - AccGood\}$ have the property:

$$\mathcal{A}(S_1) \subseteq \mathcal{N}_P^T(S_1) \cup \mathcal{P}_{nat} \bigwedge \quad (1)$$

$$\mathcal{N}_I(S_2) \subseteq \mathcal{N}_I(S_1) \bigwedge \quad (2)$$

$$\mathcal{N}_P^T(S_2) \subseteq \mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat} \quad (3)$$

By definition,

$$White(S_1) \Rightarrow (\mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1))) \bigcap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$$

From conditions (2) and (3) we have,

$$\mathcal{N}_P^T(S_2) \cup \text{Id2Prop}(\mathcal{N}_I(S_2)) \subseteq \mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat}$$

Therefore,

$$\begin{array}{rcl}(\mathcal{N}_P^T(S_2) \cup \text{Id2Prop}(\mathcal{N}_I(S_2))) & \bigcap & (\mathcal{B} \setminus \mathcal{P}_{nat}) \subseteq \\ (\mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat}) & \bigcap & (\mathcal{B} \setminus \mathcal{P}_{nat}) = \\ (\mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1))) & \bigcap & (\mathcal{B} \setminus \mathcal{P}_{nat})\end{array}$$

Hence, using definition of state whiteness we can conclude that $White(S_1) \Rightarrow White(S_2)$.
Case 2: $\{E - AccGood\}$. The rule $\{E - AccGood\}$ is

$$\frac{\begin{array}{l}\text{mp = convStrPname(m)}\\ \text{mp !< Blacklist OR mp = "bad"}\end{array}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}} \quad [\text{E}-\text{AccGood}]$$

Let $S_1 = H, l, l1[m]$ and $S_2 = H, l, l1 * mp$. The only additional property names in $(\mathcal{N}_P^T(S_2) \cup \text{Id2Prop}(\mathcal{N}_I(S_2))) \setminus (\mathcal{N}_P^T(S_1) \cup \text{Id2Prop}(\mathcal{N}_I(S_1)))$ would be the property name mp. The premise of the rule ensures that this property is not in the blacklist. Therefore $White(S_1) \Rightarrow White(S_2)$ follows in this case as well. □

We denote the set of all rules not in $\mathcal{R}^{good}$ as $\mathcal{R}^{bad}$. Finally, given a reduction trace τ, we define $\mathcal{R}(\tau)$ as the set of all axioms R_i used to derive the transitions $S_i \to S_{i+1}$ in τ (for all i). We are now ready to define the property $Pb^{strong}(S)$

Definition 11 *(Pb^{strong}) For a given state well-formed S we define $Pb^{strong}(S)$ as true iff $\mathcal{R}(\tau(S)) \subseteq \mathcal{R}^{good}$.*

The above definition basically says that a state has the property Pb^{strong} if only reduction rules from the set $\mathcal{R}^{good}$ are involved during its reduction.

Lemma 2 *For all initial states S_0 in Jb^r, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$.*

Proof. If the initial state S_0 corresponds to a value then $Pt(S_0)$ is trivially true. Therefore we consider initial states S_0 which have at least one reduction step in their trace. Let $\tau^n(S_0)$ denote the n step partial reduction trace of the state S_0, that is, $\tau^n(S_0)$ consists of the first $n + 1$ terms of the sequence $\tau(S_0)$.
In order to prove the above theorem we prove that $Pt^{strong}(S_0)$ implies the stronger property:-
$\forall n \geq 1 : \mathcal{P}(S_0, n)$ is true, where $\mathcal{P}(S_0, n)$ is defined as

$$\mathcal{A}(\tau^n(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset \quad (4)$$

$$White(S_n) \text{ is true.} \quad (5)$$

where we assume (without loss of generality) $\tau^n = S_0, S_1, \ldots, S_n$.
Clearly, for all S_0 which have at least one reduction step, $\forall n \geq 1 : \mathcal{P}(S_0, n)) \Rightarrow Pt(S_0)$.
Given that $Pt^{strong}(S_0)$ holds, we prove $\forall n \geq 0 : \mathcal{P}(S_0, n)$ by induction over n.

Base Case: $n = 1$. Let $\tau^1(S_0) = S_0, S_1$. By definition of the subset Jb^r, $White(S_0)$ holds. Since $Pt^{strong}(S_0)$ holds, the reduction rule that applies to S_0 has to be good. From goodness property of rules, $White(S_1)$ holds. From

Figure 1. List of bad reduction rules

$$\text{StrP(\_) ::= \_ in l} \mid \text{\#OPhasOwnProperty.@Exe(l1,\_)} \mid \text{\#OPpropertyIsEnumerable.@Exe(l,\_)}$$

$$\frac{\text{mp = convStrPname(m)}}{\text{H,l,StrP(m)} \longrightarrow \text{H,l,StrP(mp)}}\ [\text{E}-\text{Ctx}-\text{Str}-\text{Pname}] \qquad \frac{\begin{array}{c}\text{ParseFunction(m) = P}\\ \text{H,Function(fun(x\~{})P,\#Global) = H1,l1}\end{array}}{\text{H,l,@FunParse(x\~{},m)} \longrightarrow \text{H1,l,l1}}\ [\text{N}-\text{FunParse}-\text{StrProg}]$$

$$\frac{\text{ParseParams(m1) = x\~{}}}{\text{H,l,@FunParse(m1,m2)} \longrightarrow \text{H1,l,@FunParse(x\~{},m2)}}\ [\text{N}-\text{FunParse}-\text{StrId}]$$

$$\frac{\text{ParseProg(m) = P}}{\text{H,l,\#GEval.@Exe(l1,m)} \longrightarrow \text{H2,l,\#GEval.@Exe(l1,P)}}\ [\text{E}-\text{Eval}-\text{StrProg}]$$

$$\frac{\text{mp = convStrPname(m) AND mp} < \text{Blacklist}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}}\ [\text{E}-\text{AccBad}]$$

our semantics, for all reduction rules $\frac{(Premise)}{S_1 \to S_2}$ we know that:

$$\mathcal{A}(S_1) \quad \subseteq \quad \mathcal{N}_P^T(S_1) \cup \mathcal{P}_{nat} \tag{6}$$

Therefore $\mathcal{A}(S_0) \subseteq \mathcal{N}_P^T(S_0) \cup \mathcal{P}_{nat}$. By definition of state whiteness, it follows that $\mathcal{A}(S_0) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Hence property $\mathcal{P}(S_0, 1)$ holds.

Induction hypothesis: Assume $\mathcal{P}(S_0, n)$ is true for $n = k$. Therefore we have

$$\mathcal{A}(\tau^k(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \quad = \quad \emptyset \tag{7}$$

$$White(S_k) \text{ is true.} \tag{8}$$

Induction Step: Consider $n = k + 1$. Let $\tau^{k+1}(S_0) = S_0, S_1, \ldots, S_k, S_{k+1}$. By definition, $\mathcal{A}(\tau^{k+1}(S_0)) = \mathcal{A}(\tau^k(S_0)) \cup \mathcal{A}(S_k)$. Using condition (6) we get $\mathcal{A}(S_k) \subseteq \mathcal{N}_P^T(S_k) \cup \mathcal{P}_{nat}$. Therefore,

$$\begin{aligned}\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \quad \subseteq \quad & (\mathcal{N}_P^T(S_k) \cup \mathcal{P}_{nat}) \cap \\ & (\mathcal{B} \setminus \mathcal{P}_{nat})\end{aligned}$$

This is equivalent to

$$\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \quad \subseteq \quad \mathcal{N}_P^T(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat})$$

Since $White(S_k)$ is true $\mathcal{N}_P^T(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Therefore we have $\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Combining this with condition (7) we get

$$\mathcal{A}(\tau^{k+1}(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \quad = \quad \emptyset \tag{9}$$

From condition (8) we know that $White(S_k)$ is true. Therefore using Pb^{strong} and the goodness property of rules, $White(S_{k+1})$ is true. Combining this with condition (9), we get that the predicate $\mathcal{P}(S_0, k + 1)$ is true. Therefore $\forall n \geq 1 : \mathcal{P}(S_0, n)$ is true by induction. $\square$.

A.3.2 Step (2)

We need to show that for all initial states S_0 in the subset $Jb^r(\mathcal{B})^*$, $Pb^{strong}(S_0)$ holds. This is also the basis on which the subset $Jb^r(\mathcal{B})^*$ was obtained, that is, no term should ever move to a state where a rule from $\mathcal{R}^{bad}$ becomes applicable. We prove this property by defining a "goodness" property (inductive invariant) on heaps and terms such that: (1) For all states with a good heap and term, no reduction rule from $\mathcal{R}^{bad}$ applies. (2) Heap goodness and term goodness are always preserved during reduction.

Before defining these properties, we state a few notations. Let $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}$ denote the heap addresses of the constructor Function and methods eval, hasOwnProperty and propertyIsEnumerable of Object.prototype.

Definition 12 *(State goodness for $Jb^r(\mathcal{B})$) We say that a state S is* good, *denoted by $Good_{Jb^r(\mathcal{B})}(S)$, iff the term is good and the heap is good. The conditions for term goodness and heap goodness are given as follows.*

***Term goodness**:*

(1) *Structure of t does not contain any property name or identifier from the set $\mathcal{B} \cup \{$eval, Function, hasOwnProperty, propertyIsEnumerable constructor $\}$.*

(2) *\$ only appears inside a subexpression of the form IDX(e) for some e.*

(3) *Structure of t does not contain any sub terms with any contexts of the form eforin(), pforin(), cEval(), FunParse() contexts and any constructs of the form e in e, for (e in e) s*

(4) *Structure of t does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}$*

(5) *If Structure of t contains a sub-expression e1[e2] then for all well-formed states S such that*

$Good_{Jb^r(\mathcal{B})}(\mathcal{H}(S))$ holds and $\mathcal{T}(S) = e1[e2]$: $Pb^{strong}(S)$ is true.

Heap goodness:

$$\begin{array}{lll}
\forall l,p : H(l).p = l_{Function} & \Rightarrow & p = \textsf{constructor} \\
 & & \vee\ p = \textsf{Function} \\
\forall l,p : H(l).p = l_{eval} & \Rightarrow & p = \textsf{eval} \\
\forall l,p : H(l).p = l_{hOP} & \Rightarrow & p = \textsf{hasOwnProperty} \\
\forall l,p : H(l).p = l_{pIE} & \Rightarrow & p = \textsf{propertyIsEnumerable} \\
\forall l,p : p \in H(l) \wedge & \Rightarrow & l = l_G \\
\quad isPrefix(\$,p) & &
\end{array}$$

$isPrefix(\$,p)$ is true iff \$ is a prefix of p. The contexts pforin() and eforin are internal continuation contexts used to express the internal states obtained during the reduction of a for (e in e) s statement. FunParse() is an internal continuation context which is entered during a call to the Function constructor, in order to parse the argument string. In the rest of this Section, we will apply the predicate $Good_{Jb^r}$ to heaps, terms and states; the interpretation for each of them being the corresponding goodness definition.

Lemma 3 *For all well-formed states S_1 and S_2 in the subset $Jb^r(\mathcal{B})^*$, $S_1 \rightarrow S_2 \wedge Good_{Jb^r(\mathcal{B})}(S_1) \Rightarrow Good_{Jb^r(\mathcal{B})}(S_2)$*

Proof. We prove this lemma by an induction over the set of all reduction rules. Since state goodness holds for the initial state S_1, by Lemma 5 it is sufficient to consider only the set of good rules ($\mathcal{R}^{good}$)). All context rules which have a reduction in their premise form the inductive cases and the transition axioms form the base cases. For the base cases we prove the theorem by a detailed case analysis. For the inductive case, consider any context rule of the form $\frac{S_1 \rightarrow S_2}{C(S_1) \rightarrow C(S_2)}$ (Recall that if $S = (H, l, t)$ then $C(S) = (H, l, C(t))$). We divide the set of reduction contexts into the following three cases:

(1) C = va[−] For any state $S_1 = (H_1, l_1, t_1)$, $\mathcal{T}(C(S_1)) =$ va[t_1]. Therefore, by definition of state goodness, $Good_{Jb^r}(C(S_1))$ holds iff $Pb^{strong}(C(S_1))$ holds . By definition of Pb^{strong}, $C(S_1 \rightarrow C(S_2) \wedge Pb^{strong(C(S_1))} \Rightarrow Pb^{strong(S_2)}$. Therefore we have $Good_{Jb^r(\mathcal{B})}(C(S_1)) \Rightarrow Good_{Jb^r(\mathcal{B})}(C(S_2))$

(2) C = −[e] For any state $S_1 = (H_1, l_1, t_1), \mathcal{T}(C(S_1)) =$ t_1[e]. Therefore, by definition of state goodness, $Good_{Jb^r}(C(S_1))$ holds iff $Pb^{strong}(C(S_1))$ holds. By definition of Pb^{strong}, $C(S_1 \rightarrow C(S_2) \wedge Pb^{strong(C(S_1))} \Rightarrow Pb^{strong(S_2)}$. Therefore we have $Good_{Jb^r(\mathcal{B})}(C(S_1)) \Rightarrow Good_{Jb^r(\mathcal{B})}(C(S_2))$

(3) All other reduction contexts. For any term t, t' and an appropriate C from this set, we have the following (easy to prove) propositions.

- *Proposition 1 $Good_{Jb^r}(C(t)) \Rightarrow Good_{Jb^r}(t)$*
- *Proposition 2 $(\exists t : Good_{Jb^r}(C(t))) \wedge Good_{Jb^r}(t') \Rightarrow Good_{Jt}(C(t'))$*

From the induction hypothesis we know that
$Good_{Jb^r}(S_1) \Rightarrow Good_{Jb^r}(S_2)$. Therefore,
$Good_{Jb^r}(\mathcal{T}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{T}(S_2))$ and
$Good_{Jb^r}(\mathcal{H}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{H}(S_2))$
For all states S,
$Good_{Jb^r}(\mathcal{H}(S)) = Good_{Jb^r}(\mathcal{H}(C(S)))$. Therefore,
$Good_{Jb^r}(\mathcal{H}(C(S_1))) \Rightarrow Good_{Jb^r}(\mathcal{H}(C(S_2)))$ holds.
As a result we only need to show
$Good_{Jb^r}(\mathcal{T}(C(S_1))) \Rightarrow Good_{Jb^r}(\mathcal{T}(C(S_2)))$. This can be shown by using Propositions 1 and 2 and the induction hypothesis: $Good_{Jb^r}(\mathcal{T}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{T}(S_2))$.

□

Lemma 4 *For all well-formed expressions e_1 and e_2 in the subset Jb^r such that $Good_{Jb^r(\mathcal{B})}(e_1)$, $Good_{Jb^r(\mathcal{B})}(e_2)$ holds, and for all well-formed states S such that $Good_{Jb^r(\mathcal{B})}(\mathcal{H}(S))$ holds and $\mathcal{T}(S) =$ e1[IDX(e2)]; $Pb^{strong}(S)$ is true. In other words term goodness holds for e1[IDX(e2)]*

Proof. Let $S = (H, l, e1[IDX(e2)])$ be any state such that $Good_{Jb^r(\mathcal{B})}(H)$ holds. We need to show that $\mathcal{R}(\tau(S)) \subseteq \mathcal{R}^{good}$. According to our semantics.

$$\frac{\text{H,l,e1} \xrightarrow{*} \text{H1,l1,va1}}{\text{H,l,e1[IDX(e2)]} \xrightarrow{*} \text{H1,l1,va1[IDX(e2)]}} \quad [\text{E}-\text{eCgv}]$$

Since $Good_{Jb^r(\mathcal{B})}(H)$ and $Good_{Jb^r(\mathcal{B})}(e1)$ hold, $Good_{Jb^r(\mathcal{B})}(H, l, e1)$ is true and $Pb^{strong}(H, l, e1)$ is true. So all the transition axioms involved in $\mathcal{R}(\tau(H, l, e1))$ would be from the set of good rules. By Lemma 3, we get that $Good_{Jb^r(\mathcal{B})}(H_1)$ holds. Now we only need to argue that $\mathcal{R}(\tau(H1, l1, va1[IDX(e2)])) \subseteq \mathcal{R}^{good}$.
From our semantics we deduce that

$$\frac{\text{H1,l1,e2} \xrightarrow{*} \text{H2,l2,va2}}{\text{H1,l1,va1[IDX(e2)]} \xrightarrow{*} \text{H2,l2,va1[IDX(va2)]}} \quad [\text{E}-\text{eCgv}]$$

Again $Good_{Jb^r(\mathcal{B})}(H1)$ and $Good_{Jb^r(\mathcal{B})}(e2)$ hold, therefore $Good_{Jb^r(\mathcal{B})}(H1, l, e2)$ is true and $Pb^{strong}(H1, l, e2)$ is true. So all the transition axioms involved in $\mathcal{R}(\tau(H1, l, e2))$ would be from the set of good rules. By Lemma 3, we get that $Good_{Jb^r(\mathcal{B})}(H_2)$ holds. Now we only need to argue that $\mathcal{R}(\tau(H2, l2, va1[IDX(va2)])) \subseteq \mathcal{R}^{good}$.

This can be done using a straightforward symbolic execution based on the operational semantics rules. We deduce that $H2, l2, va1[IDX(va2)] \rightarrow H3, l3, l4 * mp$ where mp $\in \mathcal{B}$ or mp= "bad". In either case the axiom E−AccGood applies to the state and reduces it to a final value. Therefore all transition axioms involved in the entire trace are from the set of $\mathcal{R}^{good}$. Therefore $Pb^{strong}(S)$ is true. □

Lemma 5 *For all well-formed states S in the subset $Jb^r(\mathcal{B})^*$ such that $Good_{Jb^r(\mathcal{B})}(\mathcal{T}(S))$ is true, no reduction rule from $\mathcal{R}^{bad}$ applies to S.*

Proof. We prove this result by a detailed case analysis over the set of rules in $\mathcal{R}^{bad}$ and show that no rule from $\mathcal{R}^{bad}$ is applicable to any term with the term goodness property. □

Lemma 6 *For all well-formed states S_0 in $Initial(Jb^r(\mathcal{B}))$, $Good_{Jb^r(\mathcal{B})}(S_0)$ is true.*

Proof. For any initial state S_0, $\mathcal{H}(S_0)$ is the initial heap and only consists of native objects. Therefore from the semantics and the definition of heap goodness, we show that $Good_{Jb^r}(\mathcal{H}(S_0))$ holds. We show $Good_{Jb^r}(\mathcal{T}(S_0))$ by structural induction over the set of user terms in Jb^r is contained in the set of user terms Jt. The base case is straightforward. For the inductive cases, using the definition of Jb^r, we can show that conditions $(1), (2)$ and (3) in the definition of term goodness hold. Condition 4 is trivially true for all inductive cases except e1[IDX(e2)]. In this case we use Lemma 4 to argue that term goodness holds for e1[IDX(e2)]. □

Combining Lemmas 2, 3, 5, and 6 we can prove Theorem 1.

Restatement of Theorem 1 *For all well-formed states S_0 in $Initial(Jb^r(\mathcal{B}))$, $Pt(S_0)$ holds.*

Proof. From Lemma 2 we obtain, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$. Therefore proving that $Pb^{strong}(S_0)$ holds is sufficient for proving this theorem. From Lemma 5, goodness property for a state implies that no reduction rule from the set $\mathcal{R}^{bad}$ applies to it. Thus showing that for all states $S \in \tau(S_0)$, $Good_{Jb^r}(S)$ holds is sufficient to prove the theorem. From Lemma 6, $Good_{Jb^r}(S_0)$ is true and from Lemma 3, state goodness is preserved during reduction. Therefore goodness holds for all states in the trace $\tau(S_0)$. □

A.4 Proof of Claim 2

In this Subsection we prove Claim 2, which states that if P is a Firefox-JavaScript program 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 if P does not access a this bound to the scope object. We assume that the Firefox-JavaScript semantics is the ECMA-262 semantics for javascript with the difference in the rule for 'this value assignment for function calls'. In particular, the corresponding operational semantics has the rule E−CallRefAct modified to the following:

$$\frac{\text{Type(ln*m) = Reference} \quad \text{isActivation(H,ln) OR isCatch(H,ln)}}{\text{H,l,ln*m([va~])} \longrightarrow \text{H,l,@Fun(Global,ln*m[,va~])}} \text{[E-CallRefAct-mod]}$$

where the predicate isCatch(H,ln) is true iff H(ln) is the "catch-scope" object, the definition for which is elaborated in section 3.2. If P evaluates this to the scope object then Q evaluates NOSCOPE(this) to null.

We now formalize this claim in terms of the operational semantics. The this property is accessed only by the rule E−This.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)= l2}}{\text{H,l,this} \longrightarrow \text{H,l,l2}} \text{[E-This]}$$

For the sake of argument, we replace this rule by the rules below, that return null if @this points to a scope object, and the effective value of @this otherwise.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=l2} \quad \text{@scope in H(l2)}}{\text{H,l,this} \longrightarrow \text{H,l,null}} \text{[E-This-KO]}$$

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=l2} \quad \text{@scope notin H(l2)}}{\text{H,l,this} \longrightarrow \text{H,l,va}} \text{[E-This-OK]}$$

The property Ps which implies isolation of all scope objects can be formalized as follows :

Definition 13 *(Ps) Given a state S, let $S' = Final(S)$. $Ps(S)$ holds iff @Scope is not in $\mathcal{H}(S')(\mathcal{V}(S'))$.*

We denote the Firefox-JavaScript semantics along with the modification for the semantics of this as the 'modified Firefox-JavaScript semantics'.

Theorem 2 *For all well-formed states S_0 in $Initial(Js^s)$, $Ps(S_0)$ holds for execution with respect to the modified Firefox-JavaScript semantics*

In order to prove this theorem we need some supporting lemmas and definitions. As in the proof of the earlier claim, we define a goodness property on the states and show that it is inductive, and then show that the state goodness property implies the property Ps.

Definition 14 *(State goodness for Js^s) We say that a state S is good, denoted by $Good_{Js^s}(S)$, iff it has the following properties*

(1) *Structure of $\mathcal{T}(S)$ does not contain any of* eval, Function, hasOwnProperty, propertyIsEnumerable, constructor, valueOf, sort, concat, reverse. *Also $\mathcal{T}(S)$ does not contain any identifiers or property names beginning with* \$.

(2) *Structure of $\mathcal{T}(S)$ does not contain any contexts of the form* eforin(), pforin(), cEval() FunParse() *or* [ ] *contexts and any constructs of the form* e in e, for (e in e) s *and* e[e].

(3) *Structure of $\mathcal{T}(S)$ does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}, l_{valueOf}, l_{sort}, l_{concat}$ and $l_{reverse}$.*

(4) *If a heap address l is present in $\mathcal{T}(S)$ such that* @Scope *$\in \mathcal{H}(S)(l)$ is true, then l must appear inside one of the following contexts only :* Function(fun([x˜]){P},−); −.@Put(mp,va); l.@call(−,[va˜]); Fun(−,e[,va˜]); @ExeFPA(l,−,va); @FunExe(−,P); @with(−ln1,ln2,s); −∗mp.

Heap goodness

Let H denote $\mathcal{H}(S)$.

$$\begin{array}{lll}
\forall l, l', p : H(l).p = l' \wedge & \Rightarrow & p = \text{@Scope} \\
\quad \text{@Scope} \in H(l') & & \vee\ p = \text{this} \\
 & & \vee\ p = \text{@FScope} \\
\forall l, p : H(l).p = l_{Function} & \Rightarrow & p = \text{constructor} \vee \\
 & & p = Function \\
\forall l, p : H(l).p = l_{eval} & \Rightarrow & p = \text{eval} \\
\forall l, p : H(l).p = l_{hOP} & \Rightarrow & p = \text{hasOwnProperty} \\
\forall l, p : H(l).p = l_{pIE} & \Rightarrow & p = \text{propertyIsEnumerable} \\
\forall l, p : H(l).p = l_{valueOf} & \Rightarrow & p = \text{valueOf} \\
\forall l, p : H(l).p = l_{concat} & \Rightarrow & p = \text{concat} \\
\forall l, p : H(l).p = l_{sort} & \Rightarrow & p = \text{sort} \\
\forall l, p : H(l).p = l_{reverse} & \Rightarrow & p = \text{reverse} \\
\forall l, p : p \in H(l)\ \wedge \quad isPrefix(\$, p) & \Rightarrow & l = l_G \\
\$ \in H(l_G) \wedge H(l_G).\$ = true & &
\end{array}$$

$isPrefix(\$, p)$ is true iff "\$" is a prefix of the property name p. Observe that if $Good_{Js^s}(S)$ holds then there is no l such that $\mathcal{V}(S) = l$ and @Scope $\in \mathcal{H}(S)(l))$.

Lemma 7 *For all well-formed states S_1 and S_2 in the subset Js^{s*}, $S_1 \rightarrow S_2 \wedge Good_{Js^s}(S_1) \Rightarrow Good_{Js^s}(S_2)$*

Proof. We prove this lemma by an induction over the set of all reduction rules. We consider only those reduction rules that apply to good states. All context rules which have a reduction in their premise form the inductive cases and the transition axioms form the base cases. The proof is on same lines as that of lemma 4 in [**?**]. □

Lemma 8 *For all well-formed states S_0 in $Initial(Js^s)$, $Good_{Js^s}(S_0)$ is true.*

Proof. Similar to the proof for lemma 5 in [**?**]. Using the definition of Js^s, we show that $Good_{Js^s}(\mathcal{T}(S_0))$ is true. Using the semantics and the definition of heap goodness, we show that $Good_{Js^s}(\mathcal{H}(S_0))$ holds for the initial heap. □

Combining lemmas 7 and 8 we have the following proof for theorem 2.

Proof of Theorem 2 : From lemma 7, the state goodness property implies that the corresponding term can never be the address of a scope object. From lemma 8, state goodness holds for all initial states and from lemma 7, state goodness is preserved under reduction. Combining these facts, we get that all states present in $\tau(S_0)$ are good and therefore the term part for none of them would be an address of a scope object. □

We will now state and prove Theorem 3, which will show that rewriting this to NOSCOPE(this) correctly implements the rules E−This−KO and E−This−OK in terms of the original semantics with the rule E−This. These results together prove our claim.

In order for a rewritten program Q to behave exactly like the original program but with the modified firefox-JavaScript semantics we need to define \$= true in the beginning. In order to formalize this we define the subset $Initial^r(Js^s)$ as the set of states $Initial(Js^s)$ but with the heap having an additional property \$ in the global object, which is set to true.

Since Js is a subset of Jt, by Theorem 1 of [**?**], if \$ is not present as an identifier in a program then the property \$ can never be accessed. Using this it is easy to show that lemmas 7 and 8 and hence theorem 2 are true for the modified set of initial states- $Initial^r(Js^s)$.

Theorem 3 *For all states $S_1 = (H_1, l_1, t_1)$ such that $Good_{Js^s}(H_1) \bigwedge \mathcal{T}(S_1) = NOSCOPE(this)$, there exists a state $S_2 = (H_1, l_1, va)$ such that*

- *$S_1 \rightarrow^* S_2$ in the unmodified Firefox-JavaScript semantics*
- *$S_1' = (H_1, l_1, this) \rightarrow S_2$ in the modified Firefox-JavaScript semantics with rules* E−This−KO *and* E−This−OK

Proof. We prove this theorem by a symbolic execution of the semantic rules. Due to space considerations we will sketch out only the main steps of the symbolic execution. Recall that NOSCOPE(this) is the expression :

```
(this.$=false,$?(delete this.$,this):(delete this.$,$=true,null))
```

We consider the following three cases

- Case 1 : this returns the address of a scope object, say l_{scp} which is along the current scope chain. The modified Firefox-JavaScript semantics would therefore reduce S_1' to $(H_1, l_1, null)$ by rule E−This−KO.

  In the unmodified semantics,

  (1) Executing this.$= false would set the property $ of the object at l_{scp} to false. Therefore the heap after this statement would be $H_1^1 = H_1[l_{scp}.\$ = false]$.

  (2) The conditional $? in the next step would resolve to the else branch because the identifier $ would resolve to $l_{scp} * \$$ since l_{scp} shadows the global object. The heap after this statement would be $H_1^2 = H_1^1$.

  (3) Within the else branch, delete this.$ will delete property $ from object at l_{scp}. The next expression $=true will resolve to $l_G * \$ = true$ and therefore this statement will amount to setting the property $ of the global object back to true. The heap after this statement would be same as the original one, that is, $H_1^3 = H_1$.

  (4) Finally, the value null is returned and so the final state would be $(H_1, l_1, null)$.

  The final state obtained after the reduction of S_1' under the modified Firefox-JavaScript semantics is also $(H_1, l_1, null)$. Thus the theorem is true in this case.

- Case 2 : this returns the address of a non scope object, say l_o. The modified Firefox-JavaScript semantics would therefore reduce S_1' to (H_1, l_1, l_o) by rule E−This−OK.

  In the unmodified semantics,

  (1) Executing this.$= false would set the property $ of the object at l_o to false. Therefore the heap at this state would be $H_1^1 = H_1[l_o.\$ = false]$.

  (2) The conditional $? in the next step would resolve to the if branch because the identifier $ would resolve to $l_G * \$$. This is because $Good_{Js^s}(H_1)$ is true and hence for heap $H_1^1 = H_1[l_o.\$ = false]$, the only object in the current scope chain which has the property $ would be the global object. The heap after this statement would be $H_1^2 = H_1^1$.

  (3) Within the else branch, delete this.$ will delete property $ from object at l_o. The next expression $=true will resolve to $l_G * \$ = true$ and therefore this statement will amount to setting the property $ of the global object back to true. The heap after this statement would be back to the original one, that is, $H_1^3 = H_1$.

  (4) Finally, the value l_o is returned and so the final state would be (H_1, l_1, l_o).

  The final state obtained after the reduction of S_1' under the modified Firefox-JavaScript semantics is also (H_1, l_1, l_o) . Thus the theorem is true in this case.

- Case 3 : this returns the address of a scope object, say l_{scp} which is NOT along the current scope chain.

  According to the original JavaScript semantics the only case in which this can happen is when the @this property of the current activation object points to a "catch scope" object. However as explained earlier in section 3, in the Firefox-JavaScript semantics this cannot happen because of the rule E−CallRefAct−mod. Hence this case does not apply.

□

A.5 Proof of Claim 3

In this Subsection we prove Claim 3, which states that if P is a JavaScript program 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.

First of all, we need to formalize this claim in terms of the JavaScript operational semantics. The this property is accessed only by the rule E−This.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=\#Global}}{\text{H,l,this} \longrightarrow \text{H,l,null}} \quad \text{[E-This]}$$

As in the previous section, for the sake of argument, we replace this rule by the rules below, that return null if @this points to the global object, and the effective value of @this otherwise.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=\#Global}}{\text{H,l,this} \longrightarrow \text{H,l,null}} \quad \text{[E-This-KO]}$$

$$\frac{\begin{array}{l}\text{Scope(H,l,@this)=l1} \\ \text{H,l1.@Get(@this)=va} \quad \text{va!=\#Global}\end{array}}{\text{H,l,this} \longrightarrow \text{H,l,va}} \quad \text{[E-This-OK]}$$

The property Pg which implies isolation of the global object can be formalized as follows :

Definition 15 *(Pg) Given a state S, let $S' = Final(S)$. $Pg(S)$ holds iff $\mathcal{V}(Final(S)) \neq l_G$.*

We denote the ECMA-262compliant JavaScript semantics with the modification for the semantics of this as the 'modified JavaScript semantics'. By Theorem 3 of [?], we have that in modified JavaScript semantics, no program P ever evaluates to the global object.

Restatement of Theorem 3 of [?] *For all well-formed states S_0 in $Initial(Jg)$, $Pg(S_0)$ holds, under the modified JavaScript semantics.*

We will now state and prove theorem 4, which will show that rewriting this to NOGLOBAL(this) correctly implements the rules E−This−KO and E−This−OK in terms of the original semantics with the rule E−This. These results together prove our claim.

In order for a rewritten program Q to behave exactly like the original program but with the modified JavaScript semantics we need to define $= l_global in the beginning. In order to formalize this we define the subset $Initial^r(Js^{sr})$ as the set of states $Initial(Js^{sr})$ but with the heap having an additional property $ in the global object which is set to the address of the global object.

As in the previous section, we define a goodness property on the states and show that the during the execution of NOGLOBAL(this), goodness of the initial heap implies goodness of the final state. We consider the definition of states goodness $Good_{Jg}(S)$ as mentioned in definition 17 in [?]. We refine this definition by conjuncting it with two conditions :

(1) $\forall l : \$ \notin H(l)$

(2) $\mathcal{T}(S)$ does not contain any identifier or property name beginning with "$".

Since Jg is a subset of Jt, by Theorem 1 of [?], if $ is not present as an identifier in any program then the property $ can never get accessed. Using this it is easy to show that lemmas 8,9 and hence theorem 3 in [?] are true even with the refined definition of state goodness and the modified set of initial states $Initial^r(Jg)$.

Theorem 4 *For all states $S_1 = (H_1, l_1, t_1)$ such that $Good_{Jg}(H_1)$ holds and $\mathcal{T}(S_1) = NOGLOBAL(this)$, there exists a state $S_2 = (H_1, l_1, va)$ such that*

- *$S_1 \rightarrow^* S_2$ in the unmodified JavaScript semantics*
- *$S_1' = (H_1, l_1, this) \rightarrow S_2$ in the modified JavaScript semantics with rules E−This−KO and E−This−OK*

Proof. We prove this theorem by a symbolic execution of the semantic rules. Due to space considerations we will only sketch out the main steps of the symbolic execution. Recall that NOGLOBAL(this) is the expression : (this==$?null;this). $.
We consider the following two cases :

- Case 1 : this returns the address of the global object (l_G). The modified JavaScriptsemantics would therefore reduce S_1' to the state $(H_1, l_1, null)$ by rule E−This−KO.

  In the unmodified semantics

  (1) The conditional this==$? would resolve to $this == l_G * \$$. This is because the initial heap H_1 is good and therefore only the global object will have the $ property set to the address of the global object. Since this resolves to l_G, the conditional would resolve to the if branch. The heap obtained after this statement would be $H_1^1 = H_1$.

  (2) Within the if branch, the value returned would be null and therefore the final state obtained would be $(H_1, l_1, null)$.

  The final state obtained after the reduction of S_1' under the modified JavaScript semantics if also $(H_1, l_1, null)$. Thus the theorem is true in this case.

- Case 2 : this returns the address of an object, say l_o, which is different from the global object. The modified JavaScriptsemantics would therefore reduce S_1' to the state (H_1, l_1, l_o) by rule E−This−OK.

  In the unmodified semantics,

  (1) As in the previous case, the conditional this==$? would resolve to $this == l_G * \$$. This is because the initial heap H_1 is good and therefore only the global object will have the $ property set to the address of the global object. Since this resolves to l_o, the conditional would resolve to the else branch. The heap obtained after this statement would be $H_1^1 = H_1$.

  (2) Within the else branch, the value returned would be l_o and therefore the final state obtained would be (H_1, l_1, l_o).

  The final state obtained after the reduction of S_1' under the modified JavaScript semantics if also (H_1, l_1, l_o). Thus the theorem is true in this case as well.

□