Language-Based Isolation of Untrusted JavaScript

Sergio Maffeis
Department of Computing
Imperial College London
London, UK
Email: maffeis@doc.ic.ac.uk

Ankur Taly
Department of Computer Science
Stanford University
Palo Alto, USA
Email: 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 study language-based methods for filtering and rewriting JavaScript code, using Yahoo! ADSafe and Facebook FBJS as motivating examples. We explain the core problems by describing previously unknown vulnerabilities and subtleties, and develop a foundation for improved solutions based on an operational semantics of the full ECMA-262 language. We also discuss how to apply our analysis to address the JavaScript isolation problems we discovered.

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 advertising content can be placed in an isolating `iframe` [3], this is not always done because it limits the ad to a specific section of the page and prevents higher-revenue ads such as those that float over other parts of the hosting page. Similarly, social networking sites may serve untrusted content, such as applications developed by users, without isolating this content in an `iframe`. An alternative approach, explored and used by prominent Web companies, is to pre-process untrusted content, applying filters or source-to-source rewriting before the content is served. While some "JavaScript sandboxing" methods make intuitive sense, JavaScript provides many subtle ways for malicious code to subvert language-based isolation methods. These are instances of the general problem of regulating the interaction between trusted and untrusted code running in the same execution environment.

In order to provide a practically useful solution, in this paper we focus on filtering and rewriting methods for managing untrusted JavaScript [8], [10], drawing inspiration from two illustrative examples: Yahoo! ADsafe and Facebook FBJS. Facebook [22] is a leading social networking site that makes substantial use of JavaScript, allowing user-originated code to interact with trusted libraries. Yahoo! ADsafe [5] proposes a particularly flexible advertising model that supports rich interaction between advertising JavaScript code and the hosting Web page. ADsafe isolation is based on JavaScript filtering, allowing any JavaScript code that passes a static code analysis test. Facebook uses JavaScript rewriting to run applications in what is intended to be a "separate namespace" and insert certain run-time checks. While Google Caja [4] and other approaches offer alternatives, our two primary examples illustrate many core issues and provide a natural context for exploring the basic requirements for code filtering and rewriting.

We develop a formal foundation for proving isolation properties of JavaScript programs, based on our operational semantics of the full ECMA-262 Standard language (3rd Edition) [7], available on the Web [14] and described previously in [15]. We initially used this framework to prove isolation properties of ADsafe and FBJS, but in trying to do so, we discovered problems in both systems. As explained in Section 2, the version of ADsafe that was current when we started investigating it did not properly account for definitions that might occur on a hosting page, and an FBJS wrapper function could be disabled by untrusted code; both problems have since been addressed by the vendors. We also subsequently discovered that the Facebook variable-renaming process is not semantics-preserving, due to some corner cases involving properties of inherited (prototype) objects, and property names that serve as variable names when it is possible to construct a pointer to a scope object. Based on the subtlety of these errors, and others that might occur in similar systems, we believe that our detailed analysis method has significant promise as a systematic way of investigating isolation properties.

We provide a semantic basis for JavaScript filtering and rewriting by identifying sublanguages of the ECMA-262 Standard language that have certain desirable properties. Our syntactically defined subsets provide a foundation for code filtering – any JavaScript filter that only allows programs in a meaningful sublanguage will guarantee any semantic properties associated with it. We also consider subsets of JavaScript with semantic restrictions, which model the effect of rewriting JavaScript source code with "wrapper" functions. Our main technical results are 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). Because of the size of the operational semantics for the full ECMA-262 language [7], approximately 60 pages of ascii text, each of these proofs reflects significant effort. Although we have not completed a detailed study of the ways that specific browsers may depart from the ECMA-262 Standard, the properties of our subsets appears to be robust with respect to browser variations we have uncovered [16].

Related work on language-based methods for isolating the effects of potentially malicious Web content include [20], which examines ways to inspect and cleanse dynamic HTML content, and [28], which modifies questionable JavaScript, for a more restricted fragment of JavaScript than we consider here. A short workshop paper [27] also gives an architecture for server-side code analysis and instrumentation, without exploring details or specific methods for constraining JavaScript. Additional related work on rewriting based methods for controlling the execution of JavaScript include [12]. Foundational studies of limited subsets of JavaScript and dynamic languages in general are reported in [2], [25], [28], [11], [21], [1], [26]; see [16].

Plan of the paper. In Section 2, we describe FBJS, ADsafe, the vulnerabilities we discovered, and our approach for addressing the problems they raise. In Section 3, we briefly review our previous work [16] on JavaScript operational semantics. In Section 4 we use the operational semantics to identify safe subsets of JavaScript, and state their properties. The formal proofs are available in the associated technical report [17]. In Section 5, we discuss how our results can be used to address the problems found in FBJS and ADsafe, and what the vendors adopted. Concluding remarks are in Section 6.

2. JavaScript Isolation Problems

In this Section, we summarize the Facebook and ADsafe isolation mechanisms and explain some of the problems we observed with them. The Facebook vulnerabilities we describe were reported to Facebook and have been repaired. Similarly, the deficiency we observed in ADsafe was communicated to Douglas Crockford and was addressed by extending the ADsafe approach to consider properties of the hosting page.

2.1. Facebook JavaScript

Facebook [22] is a Web-based social networking application. Registered and authenticated users store private and public information on the Facebook server 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 [24], 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. For example, in the left image in Figure 1, the area in the box labelled "Alpha" is owned by the Alpha application and the "Test A" link code is written by the application publisher. 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 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 [23] 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. The design of FBJS is intended to allow application developers as much flexibility as possible, while protecting user privacy and site integrity. 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 parts 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. In fact the expression this, executed in the global scope, evaluates to the window object, which is the global scope itself. An application could

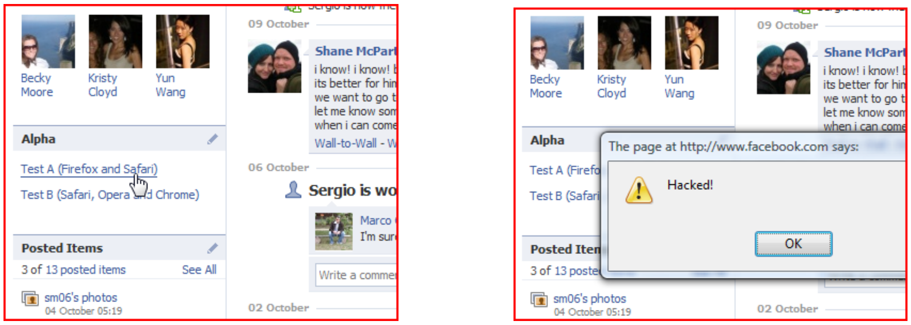

Figure 1. Demonstrating the $FBJS_{08}$ vulnerabilities in Firefox.

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 ref(this), which calls the function ref to check what object this refers to when it is used. If this refers to window, it is rewritten to null (see Section 5 for further discussion of ref and the revised version $FBJS.ref now used).

Other, indirect ways to get hold of 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[idx(e)], where idx reiterates the rewriting of blacklisted properties on the result of e. This technique is impervious to obfuscation, because 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 be used to access (indirectly) the window object, are redefined to something harmless, and is barred from using dangerous constructs such as with.

Facebook Vulnerabilities Found. We initially attempted to use our operational semantics of JavaScript [14] to prove that the subset of JavaScript used in FBJS has certain semantic properties that provide meaningful isolation between an FBJS application and the enclosing Facebook page, in particular restricting access to the window object. In the process, we uncovered certain problem cases that led to discovery of vulnerabilities in the then-current version of FBJS (see Figure 1). When we contacted Facebook, these vulnerabilities were repaired within 24 hours. For simplicity, we refer to the Facebook isolation mechanisms that were current in late 2008 as $FBJS_{08}$.

Library Leaks. A necessary condition for the safety of FBJS is that no predefined library function that is exposed to the untrusted code should return anything dangerous, in particular the window object. Analyzing the $FBJS_{08}$ libraries, we found two methods that returned their this: setSendSuccessHandler of LiveMessage.prototype and htmlEncode of String.prototype. If we extract one of these methods from its respective objects, and call it as a stand-alone function, we obtain the window object, as specified by the operational semantics of JavaScript.

Altering the Scope. A more interesting and significant discovery was that the run-time monitoring functions ref and idx could be switched off, due to a semantic subtlety of JavaScript. The nature of these vulnerabilities can be understood by assuming that $FBJS_{08}$ programs can contain an expression get_scope() which returns the current scope object; two ways of achieving this are explained below. Once a program has a handle to its own scope object, the $FBJS_{08}$ run-time checks could be disabled by replacing the ref or the idx functions, such as by running

```
get_scope().ref=function(x){return x}
```

With the run-time-checking function out of the way, ref(this) returns the current value of this, even when it is the window object.

One way to define get_scope() so that it returns the current scope object is by the code

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

In $FBJS_{08}$, this code is rewritten to

```
try {throw ( function(){return ref(this)} );}
catch (a12345_get_scope){...}
```

When the code is executed, the function thrown as an exception in the try block is bound to the identifier a12345_get_scope in a new scope object that becomes the scope for the catch block. If we execute within the catch block the function call a12345_get_scope(), the this identifier of the function is bound to the enclosing scope object. But the Facebook run-time monitor ref lets the scope object (which is different from the window object) be returned by the a12345_get_scope function, enabling the attack described above. In fact, the scope object looks exactly like any other innocuous object to the ref function.

```
<a href="#" onclick="a()">Test A (Firefox and Safari)</a>
<script>var get_win = function get_scope(x){
              if (x==0) {return this}
              else {get_scope(0).ref=function(x){return x};
                     return get_win(0)}};
function a(){get_win(1).alert("Hacked!")}</script>
```

```
<a href="#" onclick="b()">
Test B (Safari, Opera and Chrome)</a>
<script>function b(){
  try {throw (function(){return this});}
  catch (get_scope){get_scope().ref=function(x){return x};
                     this.alert("Hacked!")}}</script>
```

Figure 2. $FBJS_{08}$ exploit code.

There is another, even more subtle way to access the scope object, by the recursive code

```
var get_window =
    function get_scope(x){
        if (x==0) {return this}
        else {...get_scope(0)...}
    }
```

Here we save a named function in a global variable. As this function executes, the static scope of the recursive function is a fresh scope object o where the identifier get_scope is bound to the function itself, making recursion possible. If we invoke get_window(1) and in the else branch we recursively call get_scope(0), then this latter function call gets the this bound to the scope object o mentioned above. Such object escapes the ref check, and can be returned by the recursive call get_scope(0), and used to disable refand escape from the sandbox as described above (the full code is reported in Figure 2).

Demonstrating the Vulnerabilities. Access to the window object gives a $FBJS_{08}$-application-based attacker control over the whole Facebook page. The privileges obtained by the attacker include reading the page `cookie`, altering the user profile, interfering with other Facebook applications, suppressing advertisement and exploiting potential browser vulnerabilities. See Felt *et al.* [9] for discussion of further ramifications of the exploit. In Figure 2 we give JavaScript for the $FBJS_{08}$ attacks involving the scope described above. It simply opens an unauthorized pop-up dialog (screen shots are in Figure 1).

The two vulnerabilities due two library leaks could be exploited in all JavaScript enabled browsers. The effectiveness of the scope-related attacks instead is browser-dependent because of deviations from the ECMA-262 specification. Since Safari follows the specification in handling both the try-catch construct and recursive functions, it is vulnerable to both attacks. Opera and Chrome follow the try-catch specification but depart from it on the recursive function by binding the window object instead of the scope object to this. Hence they are vulnerable to attack B only. Firefox does the opposite, binding window to this in the try-catch case, and following the specification in the recursive function case. Hence, it is vulnerable to attack A only. Internet Explorer 7, as tested, departs from the specification binding window to this in both cases, and is therefore not vulnerable to these specific attacks.

2.2. Safe Advertising with ADsafe

Many Web pages display advertisements, which typically are produced by untrusted third parties (online advertising agencies) unknown to the publisher of the hosting page. Even an ad as simple as an image banner is often loaded dynamically from a remote source by running a piece of JavaScript provided by the advertiser or some (perhaps untrusted) intermediary. Hence, it is important to isolate Web pages from advertising content, which may potentially consist of a malicious script. As mentioned earlier, an advertisement may be placed inside an HTML `iframe`, which is isolated according to the browser same-origin policy [3].

The ADsafe JavaScript subset proposed by Yahoo! is designed to allow advertising code to be placed directly on the host page, limiting interaction by a combination of static analysis and syntactic restrictions. As explained in the documentation [5], "*ADsafe defines a subset of JavaScript that is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by tools like JSLint so that no human inspection is necessary to review guest code for safety.*". The high–level goal of ADsafe is to "*block a script from accessing any global variables or from directly accessing the DOM or any of its elements*". The advertising code has instead access to an ADSAFE object, provided as a library, that mediates access to the DOM and other page services. For example, the JavaScript code

```
var location = document.location
```

that accesses the DOM, should be written by the user as

```
var location = ADSAFE.get(document,"location")
```

Access to user-defined objects does not need to be mediated by the ADsafe wrappers, as in

```
var o={l:0}; o.l=42
```

Using our JavaScript operational semantics [14], [15], we tried to prove that the 2007 version of ADsafe [6] indeed isolated ADsafe-compliant JavaScript code from the global variables (that is, the properties of window). In setting up the proof, however, we found a problem with the ADsafe design:

the page hosting a ADsafe-compliant advertisement may unwittingly define objects or add properties to accessible objects in a way that provides access to the global scope. If the page hosting an advertisement adds a dangerous function f to Object.prototype, then the ADsafe-compliant code

```
var o={};o.f()
```

is able to call f (because o inherits from Object.prototype), and potentially violate the intended isolation properties.

In fact, we found that a very common JavaScript library, `prototype.js` [19], provides ways for ADsafe-compliant code to access the global scope. For example, an eval method is added to String.prototype, allowing arbitrary code computed by string manipulation to be executed. We notified the authors of ADsafe about this problem, which has since been addressed by imposing restrictions on any page hosting an ad (see Section 5). However, these restrictions are not specified with the same precision as other ADsafe guidelines, leading us to believe that further investigation is warranted.

2.3. Formalizing JavaScript Isolation

The FBJS and ADsafe examples above illustrate 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 that satisfies a specific syntactic criterium 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. Since even organizations that have devoted significant time and effort to deploying such language-based mechanisms have overlooked certain problems (as illustrated by the attacks above), we believe that a fundamental study based on traditional programming language foundations to design *provably secure* isolation techniques is needed. As a first step, we set up to define syntactic subsets of JavaScript that enforce isolation, and prove that they indeed do so.

3. JavaScript Semantics

In this Section we briefly summarize our formalization of the operational semantics of JavaScript [14], [15] based on the ECMA-262 standard [7], and introduce some auxiliary notation and definitions. In [16], 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 of isolation properties considered in Section 4.

Browser implementations of JavaScript extend the standard by providing additional reflection mechanisms, and most notably the DOM libraries to interact with the browser window. Mostly, these extension can be considered as an additional set of native JavaScript objects and functions preloaded in memory, and do not affect the overall definition of the operational semantics. Further discussion of the relation between this semantics and current browsers implementations appears in [16].

3.1. Operational Semantics

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

Syntactic Conventions. We abbreviate t1,..., tn with t˜ and t1 ... tn with t∗ (t+ in the nonempty case). In a grammar, [t] means that t is optional, t|s means either t or s, and in case of ambiguity we escape with apices, as in escaping [by *"["*. Internal values, which are used only in the semantics and are not part of the user syntax, are prefixed with &, as in &NaN. For conciseness, we use short sequences of letters to denote metavariables of a specific type. For example, m ranges over strings, pv over primitive values, etc.. These conventions are summarized in Figure 3.

Heaps and Values. Heaps map locations to objects, which are records of pure values va or functions fun(x,...){P}, indexed by strings m or internal identifiers @x (the symbol @ distinguishes internal from user identifiers). Values are standard. As a convention, we append w to a syntactic category to denote that the corresponding term may belong to that category or be an exception. For example, lw denotes an address or an exception. We assume a standard set of functions to manipulate heaps. alloc(H,o) = H1,l allocates o in H returning a fresh address l for o in H1. H(l) = o retrieves o from l in H. o.i = va gets the value of property i of o. o−i = fun([x˜]){P} gets the function stored in property i of o. o:i = {[a˜]} gets the possibly empty set of attributes of property i of o. H(l.i=ov)=H1 sets the property i of l in H to the object value ov. del(H,l,i) = H1 deletes i from l in H. i !< o holds if o does not have property i. i < o holds if o has property i.

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.

```
H ::= (l:o)˜ % heap
l ::= #x % object addresses
x ::= foo | bar | ... % identifiers
o ::= ”{”[(i:ov)˜]”}” % objects
i ::= m | @x % indexes
ov ::= va[”{”a˜”}”] % object values
        | fun”(”[x˜]”){”P”}” % function
a ::= ReadOnly| DontEnum | DontDelete % attributes
pv ::= m | n | b | null | &undefined % primitive values
m ::= ”foo” | ”bar” | ... % strings
n ::= −n | &NaN | &Infinity | 0 | 1 | ... % numbers
b ::= true | false % booleans
va ::= pv | l % pure values
r ::= ln”∗”m % references
ln ::= l | null % nullable addresses
v ::= va | r % values
w ::= ”<”va”>” % exception
t ::= P | s | e % terms: program, statements and expressions
```

Figure 3. Syntax for Values and Meta-Variables.

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 as set membership, inequality and semantic function application.

An atomic transition is described by an axiom. For example, the axiom $\text{H,l,(v)} \longrightarrow \text{H,l,v}$ 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.

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

The full formal semantics [14] contains several other contextual rules to account for other mutual dependencies and for all the implicit type conversions. This substantial use of contextual rules greatly simplifies the semantics and will be very useful in Section 4 to prove its formal properties.

Scope and Prototype Lookup. The scope and prototype chains are two distinctive features of JavaScript. The stack is represented by a chain of objects whose properties represent the binding of local variables in the scope. Since we are not concerned with performance, our semantics needs to know only a pointer to the head of the chain (the current scope object). Each scope object stores a pointer to its enclosing scope object in an internal @Scope property. This helps in dealing with constructs that modify the scope chain, such as function calls and the with expression.

JavaScript follows a prototype-based approach to inheritance. Each object stores in an internal property @Prototype a pointer to its prototype object, and inherits its properties. At the root of the prototype tree there is @Object.prototype, that has a null prototype. The rules below illustrate prototype chain lookup.

$$\text{Prototype(H,null,m)=null}$$

$$\frac{\text{m!} < \text{H(l)} \quad \text{H(l).@Prototype=ln}}{\text{Prototype(H,l,m)=Prototype(H,ln,m)}} \qquad \frac{\text{m} < \text{H(l)}}{\text{Prototype(H,l,m)=l}}$$

Function Scope(H,l,m) returns the address of the scope object in H that first defines property m, starting from the current scope l. It is used to look up identifiers in the semantics of expressions. Its definition is similar to the one for prototype, except that the condition (H,l.@HasProperty(m)) (which navigates the prototype chain to check if l has property m) is used instead of the direct check m < H(l).

Types. JavaScript values are dynamically typed. Types T∈ {Undefined,Null,Boolean,String,Number,Object,Reference} are used to determine conditions under which certain semantic rules can be evaluated. The semantics defines simple predicates and functions which perform useful checks on the type of values.

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. The syntax for user expressions is reported in Figure 4, where we use &PO,&UN,&BIN to range respectively over primitive, unary and binary operators.

The semantics of most user expressions is similar to usual programming languages such as Java, but some expressions are particularly subtle in JavaScript. While an in-depth description of all corner cases goes beyond the scope of this paper, we can highlight a few of them to illustrate the difficulty of dealing with JavaScript, and address the reader to [15], [10] for additional details.

For example, the expressions 1 == ”1” evaluates to true, because the == operator converts its arguments to have the same type before testing for equality between basic

```
e ::=
      this % the ”this” object
      x % identifier
      pv % primitive value
      ”[” [e˜] ”]” % array literal
      ”{”[(pn:e)˜]”}” % object literal
      ”(”e”)” % parenthesis expression
      e.x % property accessor
      e”[”e”]” % member selector
      new e[”(”[e˜]”)”] % constructor invocation
      e”(”[e˜]”)” % function invocation
      function [x] ”(”[x˜]”){”[P]”}” % [named] function expression
      e &PO % postfix operator
      &UN e % unary operators
      e &BIN e % binary operators
      ”(”e”?”e”:”e”)” % conditional expression
      (e,e) % sequential expression

pn ::= n | m | x % property name
```

Figure 4. Syntax for Expressions

values (of course, 1 == "2" evaluates to false). A more cryptic example is the expression below, that evaluates to 42:

```
(f = function(){},
 f.prototype = {a:12},
 o = new f,
 o.toString = function(){return 30},
 o["a"] + o)
```

The code creates an empty function f, creates a prototype property on f and assigns to it an object where a contains 12. The next line generates an object o (using f as a constructor) which has the prototype as described above, and the last line accesses the property a inherited by o through the prototype chain, and calls implicitly o.toString, yielding the result of 12+30. As a last example, the expression f.constructor yields the original content of the global variable Function, which is a predefined constructor for functions. The expression f = function(){} is equivalent to f = new Function, but only if Function has not been redefined to something else by some user code. Hopefully, these examples give a taste of the subtlety of the highly reflective JavaScript semantics.

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
```

```
s ::=
      ”{”s∗”}” % block
      var [(x[”=”e])˜] % assignment
      ; % skip
      e % expression not starting with ””,”function”
      if ”(”e”)” s [else s] % conditional
      while ”(”e”)” s % while
      do s while ”(”e”)”; % do-while
      for ”(”e in e”)” s % for-in
      for ”(”var x[”=”e] in e”)” s % for-var-in
      continue [x]; % continue
      break [x]; % break
      return [e]; % return
      with ”(”e”)” s % with
      id:s % label
      throw e; % throw
      try ”{”s∗”}” [catch ”(”x”){”s1∗”}”] [finally ”{”s2∗”}”] % try

P ::= fd [P] | s [P]

fd ::= function x ”(”[x˜]”){”[P]”}”
```

Figure 5. Syntax for Statements and Programs

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.

The user statements are reported in Figure 5. Their semantics is mostly standard, and we address the reader once again to [15], [10] for additional details. In Section 2 we discussed some subtle aspects of the semantics of the try−catch statement.

Programs. Programs are sequences of statements and function declarations (Figure 5. As usual, the execution of statements is taken care of by a contextual rule. Evaluating a statement to a break or continue outside of a control construct raises an exception:

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

The run-time semantics of a function declaration instead is equivalent to a no-op:

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

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

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

which adds to the initial heap NativeEnv first the variable and then the function declarations (functions VD,FD).

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.

Eval. The eval function takes a string and tries to parse it as a legal program text. If it fails, it throws a SyntaxError exception (rule omitted). If it succeeds, it parses the code for variable and function declarations (respectively VD,FD) and spawns the internal statement @cEval (rule omitted).

Object. The @Object constructor is used for creating new user objects and internally by constructs such as object literals. Its prototype @ObjectProt becomes the prototype of any object constructed in this way, so its properties are inherited by most JavaScript objects. @ObjectProt is the root of the scope prototype chain and, its internal prototype is null. Apart from *"constructor"*, which stores a pointer to @Object, the other public properties are native meta-functions such as toString or valueOf (which, like user functions, always receive a value for @this as the first parameter).

3.2. Preliminaries

We now define some notation and state some properties of the semantics that support the formal analysis of JavaScript subsets of 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 Appendix A for a formal definition). In [15], 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 *reduction trace* τ is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S_1 \to \ldots \to S_n \to \ldots$. Given 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 all 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.

$$\begin{aligned}
&\mathcal{N}_I^T(S) = \{x | x \in \mathcal{T}(S)\}\\
&\mathcal{N}_P^T(S) = \{mp \mid mp \in \mathcal{T}(S)\}\\
&\mathcal{N}_I^H(S) = \{x \mid x \in P,\ P \in \mathcal{H}(S)\}\\
&\mathcal{N}_P^H(S) = \{mp \mid \exists l : mp \in \mathcal{H}(S)(l)\}\\
&\mathcal{N}_I(S) = \mathcal{N}_I^T(S) \cup \mathcal{N}_I^H(S)\\
&\mathcal{N}_P(S) = \mathcal{N}_P^T(S) \cup \mathcal{N}_P^H(S)\\
&\mathcal{N}(S) \ = \ \mathcal{N}_I(S) \ \cup \ \text{Prop2Id}(\mathcal{N}_P(S))
\end{aligned}$$

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)$\}$, (using a prefix notation for the functions defined in Section 3.1).

Definition 1: (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 section 4, we will consider syntactic subsets of JavaScript. Unless we specify otherwise, a syntactic subset J will always denote 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.

4. Secure JavaScript Subsets

In this Section, we propose secure subsets of JavaScript and prove their formal properties. As described in Section 2, the ultimate goal is to make sure that a piece of code written in the safe subset does not access certain global variables. Those variables may contain libraries with privileged functions, or may simply belong to the name space of a different piece of code coming from another application. One approach to achieve this is to enforce exclusively syntactic restrictions, so that the user code that belongs to a safe subset is directly executed in the browser. An alternative approach is to complement syntactic restrictions with the insertion of run-time checks to monitor user code at run time (such as ref and idx in FBJS). The first approach is more efficient, more robust with respect to JavaScript code introspection and guarantees that the semantics of the user code is unaltered. The second approach is more flexible, resulting in larger subsets of JavaScript, but introduces run-time overhead and may give raise to unexpected run-time errors. In this paper, we focus on the first, purely syntactic approach. A formal analysis of run-time checking requires different analysis techniques, and we leave it to future work.

Three JavaScript Subsets In order to isolate global variables, we need to solve a crucial problem: determine the set of properties that a piece of code can access.

Our first subset, Jt, is designed to solve this problem without restricting the use of this. In Section 2, we have seen how a JavaScript program can get hold of the scope by way of this. Manipulating the scope leads to a confusion of the boundary between variables (which are properties of scope objects) and properties of regular object. For example, the expression

```
var x; this.x=42
```

effectively assigns 42 to variable x. Hence, Jt code cannot use as property name any of the global variable names to be protected. In theory, this does not constitute a significant limitation of expressiveness. In fact, 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).

For several practical purposes, forbidding the usage of this is too restrictive. In fact, this is important for object-oriented behaviour in JavaScript. To reinstate this, we need to solve the problem of isolating the window object, hence the global scope. Our last subset, called Jg, is defined for a hypothetical JavaScript semantics that forbids this to be bound to window. Since the local scope of try-catch blocks and recursive functions can still be directly manipulated, in general variables can still be confused with property names, and therefore variable renaming does not preserve the meaning of programs. Yet, this difference can be observed only in unusual corner cases. On the other hand, since variables defined in the global scope *are* effectively separated from property names, this subset can still be used to isolate the namespaces of different applications just like in Js.

4.1. Isolating property names: Jt

The first technical problem we consider is to determine the set of property names that may be accessed by a piece of code. This problem is intractable for JavaScript in general, because property names can be computed using string operations, as in

```
var o = {prop:42}; var m = "pr"; var 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 [15].

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

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

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

We now formalize the property that if the execution of a program P accesses the property p of some object, then either $p \in \mathcal{P}_{nat}$ or p appears textually in P, expressed here using $\mathsf{Id2Prop}(\mathcal{N}_I^T(S))$ to convert to property names the identifiers appearing in the term of state S.

Definition 3: (Pt) Given a state S, $Pt(S)$ holds iff

$$\mathcal{A}(\tau(S)) \subseteq \mathsf{Id2Prop}(\mathcal{N}_I^T(S)) \;\cup\; \mathcal{P}_{nat}.$$

To violate this condition, a program must access a property name generated by the conversion of a string to a piece of code. Thus, to identify all terms which lead to the execution of reduction rules for converting string to code or property names, and we remove them from Jt.

Definition 4: 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.

Our definition of property access includes checking for the existence of a property. Therefore, in order to guarantee this property 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 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]. In most cases eval is used to simplify the parsing of JSON messages or to obfuscate code. Its use is not strictly necessary for the majority of web applications, except the malicious ones, which rely heavily on obfuscation. In fact, eval is commonly considered *evil*, and is excluded from most practical subsets. The member access notation constitutes the natural way to access array elements. Arrays, and iteration over their elements can still be used in Jt, by replacing the assignment expression a[n]=e, where a is an array and n a number, by a.splice(n,1,e), and the reference expression a[n] by a.slice(n,n+1).pop(). For example, var x = a[n] becomes var x = a.slice(n,n+1).pop(). While this translation introduces a performance overhead and may annoy a programmer, it shows that the expressiveness of our subset, and therefore its usefulness, is not seriously hampered. As mentioned in Section 2, an alternative to removing e[e] (that we plan to investigate in future work) is to insert a run-time check on the argument e[idx(e)].

Theorem 1: For all well-formed states S_0 in $Initial(Jt)$, $Pt(S_0)$ holds.

Theorem 1 implies that 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 stored in the heap. A simple static analysis can be used to screen the actual code for blacklisted properties. Since the initial JavaScript heap is defined by the specification, blacklisting can be effectively enforced as long as the code of any user-defined function pre-loaded in the heap is known *a priori* (such is the case for Facebook).

4.2. Protecting the Scope: Js

We now consider a subset that keeps variables distinct from property names by preventing manipulation of explicit scope objects. In order to do so, we must prevent any user expression to evaluate to a scope object. Scope objects of course can still be accessed implicitly by the internal semantics steps corresponding to the resolution of identifiers and the creation of functions, otherwise the language would be useless.

Let $\mathcal{V}$ be a function that returns the value of a final state, and null otherwise. That is, $\mathcal{V}(S) =$ vae if $\mathcal{T}(S) =$ vae or (ct, vae, xe), and $\mathcal{V}(S) =$ null otherwise. We define a property Ps which implies that no user-defined expression can evaluate to a scope objects.

Definition 5: (Ps) Given a state S, let $S' = Final(S)$. $Ps(S)$ holds iff @Scope is not in $\mathcal{H}(S')(\mathcal{V}(S'))$.

Note that this definition is not restrictive, in the sense that any state such that $\mathcal{V}(S) \neq$null in necessarily a final state.

Combining Ps with the property Pt, described in Section 4.1, we obtain the subset Js which isolates scope objects.

Definition 6: The subset Js is defined as Jt minus all terms containing this, with(e){s} and the identifiers valueOf, sort, concat and reverse.

First, the subset forbids any use of this, which can be used to access the scope as detailed in Section 2. Just like in FBJS, we need to remove 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 the internal properties @Scope, @FScope and @this, which in turn can only be accessed as a side effect of the execution of other instructions. For example, the @Scope property is accessed during identifier resolution, in order to search along the scope chain. However, the contents of the @Scope property are never returned as the result of a reduction step. The same is true for @FScope, which denotes the scope pointer of a function closure. The @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.

Theorem 2: For all well-formed states S_0 in $Initial(Js)$, $Ps(S_0)$ holds.

Theorem 2 gives a strong safety guarantee on Js. As we shall see in Section 4.4, Js is the only JavaScript subset (among the ones considered in this paper) where renaming can be completely transparent.

4.3. Isolating the Global Object: Jg

In Js, we exclude this because it can be used to obtain a scope object. However, there are common object-oriented programming patterns when the @this property of the current scope object does not contain a scope object and therefore can be used safely. For example, in the code below, this is bound to object o during the execution of o.getval().

```
var o = {val:10, getval:function(){return this.val}};
o.getval()
```

Disallowing this altogether would break many existing JavaScript libraries, and entail extensive rewriting. We consider instead a weaker property, saying that no user expression can evaluate to the global scope. Of course the global object is still accessed implicitly during a computation, for example when resolving a global identifier.

Definition 7: (Pg) Given a state S, $P_{global}(S)$ holds iff $\mathcal{V}(Final(S)) \neq l_G$.

As a counterpart to the run-time checking technique used by FBJS to monitor the actual value of this, we define an alternative semantics for JavaScript where the window object is never returned by a this expression.

$$\frac{\begin{array}{c}\text{Scope(H,l,@this)=l1} \qquad \text{H,l1.@Get(@this)=va}\\ \textit{IF}\ \text{va = l\_global}\ \textit{THEN}\ \text{ln = null}\ \textit{ELSE}\ \text{ln = va}\end{array}}{\text{H,l,this} \longrightarrow \text{H,l,ln}}$$

Assuming that our alternative semantics can be correctly implemented by run-time checks, we define a subset that allows this yet keeps global variables separate from generic property names, and therefore support flexible isolation policies, just like Js.

Definition 8: The subset Jg is defined as Jt minus all terms containing identifiers valueOf, sort, concat and reverse.

Jg includes both this and with. It includes with because the expression with(e){s}, that alters the scope of s by adding e on top of the scope chain, does not provide a new way to obtain the window object.

On the other hand, Jg still excludes the native functions valueOf, sort, concat and reverse because they return window, if called in the appropriate context. An alternative would be to allow such functions, and define an alternative semantics for them that returns null instead of window. We do not follow this approach because such a semantics would be hard, if not impossible, to enforce in practice.

Theorem 3: For all well-formed states S_0 in $Initial(Jg)$, $Pg(S_0)$ holds.

4.4. Closure under renaming

The final technical problem we consider is the ability to rename variables in JavaScript code. Variable renaming is difficult for full JavaScript, because property names (and therefore variable names, which are properties of a scope object) may be computed by string operations, and scope objects can be explicitly manipulated. However, we are going to show that the subset Js, which prevents both cases, fully supports variable renaming.

The goal of variable renaming is to isolate the namespaces of different applications without requesting 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 very subtle. In particular, we should not rename all variables corresponding to native properties of any scope object, including the ones inherited via the prototype chain. Those properties in fact have a predefined semantics that cannot be preserved by renaming. The most obvious example is the expression toString(), that evaluates to *"[object␣Window]"*, whereas raises a reference error exception when it is evaluated as a12345_toString() in the renamed version.

Since Js does not contain with, the only things that can be scope objects are the global object, internal activation objects or freshly allocated objects (in the case of try-catch and named functions). Therefore 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:

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

A browser implementation will contain additional properties such as document,setTimeout,etc..

Recall from Section 3.2 that given a state S, $\mathcal{N}(S)$ denotes the set of all possible names appearing in S.

Definition 9: Given a state S, a partial injective function α from identifiers to identifiers is a *safe renaming for S* iff $dom(\alpha) \cap \mathcal{P}_{noRen} = \emptyset$, and $\forall x \in dom(\alpha) : \alpha(x) \notin \mathcal{N}(S)$.

The last condition means that α introduces only names *fresh* with respect to state S. A safe renaming is applied to a state S by: renaming the formal parameters and the body of all the user functions stored in $\mathcal{H}(S)$; renaming all the properties of scope objects in $\mathcal{H}(S)$; renaming all identifiers in $\mathcal{T}(S)$; renaming all the property names occurring in $\mathcal{T}(S)$ inside a particular set of contexts for internal terms. The formal definition is in Appendix B. (The definition extends to traces in the obvious way). Note that if $\mathcal{H}(S_0)$ is the initial heap with no user code then $\alpha(\mathcal{H}(S_0)) = \mathcal{H}(S_0)$, and the names of the initial state $\mathcal{N}(S_0) = \mathcal{N}_I^T(S_0) \cup \mathcal{N}_P^0$, which can be determined by a simple syntactic inspection of the code.

Assuming this definition of variable renaming, we have that the intended meaning of a Js program does not change under renaming.

Theorem 4: For all well-formed states S_0 in $\mathit{Initial}(Js)$, if α is a safe renaming function with respect to S_0, then $\alpha(\tau(S_0))$ equals $\tau(\alpha(S_0))$.

On the other hand, Jt and Jg do not support the semantics preserving renaming of variables. The counterexample

```
try {throw (function(){return this});}
catch(y){y().x=42; x;}
```

is valid Jt and Jg 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.

5. Applications: FBJS and ADsafe

In this Section, we explain how the results about subsets of ECMA-262 Standard JavaScript proved in Section 4 can be used to address the ADsafe and FBJS isolation problems explained in Section 2. We also compare our semantics-supported suggestions to the repairs that FBJS and ADsafe adopted in response to our disclosures to them.

As noted earlier, specific browsers may implement versions of JavaScript that extend the ECMA-262 Standard or differ from it in certain ways. The most striking differences lie in support for user-defined "getters" and "setters", which allow user code to redefine the way a property p of object o is read or written when the "dot" notation o.p is used. In addition, browsers provide DOM objects and may support syntactic extensions. (Examples appear in [16].) In principle, for browsers that support a variant of the ECMA-262 Standard, our results on subsets of JavaScript may be applied by further restricting the subset to eliminate places where the browser implementation is at variance with the standard. In practice, FBJS and ADsafe forbid all property names beginning with "__", which prohibits extensions such as getters and setters, and provide wrapper functions to limit the usage of DOM objects. However, we leave detailed analysis of (i) browser variants of the ECMA-262 Standard and (ii) semantic proofs of the effectiveness of wrapper functions and other dynamic checks to future work.

5.1. Fixing FBJS

Within hours of our disclosure to them, the Facebook team addressed the problems discussed in Section 2. The team fixed the library leaks associated with setSendSuccessHandler and htmlEncode by adding a check that this is different from window. To fix the scope problem, they separated the namespace of the run-time checks ref and idx from the namespace available to FBJS applications, by adding the two functions as properties of a private object $FBJS, and preventing user code from using $FBJS as a property name. This thwarts the attacks reported in Figure 2 because an expression like get_scope().$FBJS is rewritten to a12345_get_scope().__unknown__.

The FBJS isolation problem is to prevent the code of untrusted applications to access certain blacklisted global variables, and to interfere directly with each another. If two separate sections of JavaScript code use an undeclared variable x, this will be treated as the same global variable in both of them. To keep code in one Facebook application from interfering with code in another through such a variable, the Facebook site renames variables in each application by adding an application-specific prefix, as discussed in Section 2.

A purely syntactic solution to the FBJS isolation problem, justified by our analysis, is to restrict Facebook applications to our subset Js. This could be an attractive solution for isolating user-supplied applications in contexts where code is written from scratch, so that it can avoid to use the this. By Theorem 4, we can separate the namespaces of different applications, and of the FBJS libraries, without altering their semantics. By Theorem 1, a simple syntactic check on application code guarantees that it cannot escape its namespace or access blacklisted properties.

An alternative solution, closer in spirit to FBJS, is to use the subset Jg, and blacklist the $FBJS global variable so that it cannot appear in user code (Theorem 1). Informally, we can argue that the alternative semantics of this assumed in the definition of Jg is implemented by the $FBJS.ref(this) check. (In future work, when we study run-time checks for JavaScript, we plan to justify this statement formally.) By Theorem 3, the global object cannot be accessed, yet application code can freely use this. By Theorem 1, a simple syntactic check on application code guarantees that it cannot access blacklisted properties. However, as discussed in Section 4.4, there are some subtleties involving renaming, because this lets user code manipulate scope objects directly.

Besides proposing constructive solutions to the FBJS isolation problem, our semantic analysis let us discover real problems in the deployed Facebook platform. The vulnerabilities of Section 2 are a direct consequence of the fact that $FBJS_{08}$ did not implement correctly the alternative semantics of this. In particular, besides omitting to sanitize certain library functions, $FBJS_{08}$ did not blacklist ref. Moreover, we discovered two ways in which the renaming discipline adopted by the current version of FBJS, does not preserve the semantics of user programs. FBJS programs can manipulate their scope (at least in some browsers) and the FBJS renaming is not a *safe renaming* in the sense of Section 4.4 because it renames properties in $\mathcal{P}_{noRen}$, such as Object, which are hard-wired in the JavaScript semantics. Therefore, to achieve semantics-preserving renaming, FBJS should be further restricted to prohibit these names (or provide a faithful emulation for each of them), and $FBJS.ref should not return scope objects.

5.2. Enforcing ADsafe

Shortly after we notified Yahoo! of the problems described in Section 2, the ADsafe [5] documentation was amended with an additional constraint that "*None of the prototypes of the built-in types may be augmented with methods that can breach ADsafe's containment*". This is only a partial solution in that it requires the editor of the hosting page to make sure that a fairly complicated requirement is satisfied, without providing specific guidance on how to do so.

We propose a different approach. The page that agrees to safely host an ADsafe advertisement must provide two list of "dangerous" property names $\mathcal{P}_{noW}$ and $\mathcal{P}_{noRW}$, such that all illicit accesses to blacklisted properties (or this) arise from either writing to a property in the set $\mathcal{P}_{noW}$ or reading or writing to a property in $\mathcal{P}_{noRW}$. For example, the set $\mathcal{P}_{noW}$ may include the native properties toString, toSource and those in $\mathcal{P}_{nat}$. The set $\mathcal{P}_{noRW}$ includes by default security-critical properties such as eval, window, cookie, and the other properties and methods that can be invoked to reach these. We have not developed an analysis method to make the generation of these black list automatic, but it may be possible to do so using call-graph analysis.

The admissible ADsafe code for a hosting page is taken as a subset of Js, after filtering out all adds mentioning the blacklisted properties in $\mathcal{P}_{noW}$ or $\mathcal{P}_{noRW}$. The soundness of this approach follows from Theorem 1 and Theorem 3. The severity of syntactically restricting advertisements depends on the nature of the sets $\mathcal{P}_{noW}$ and $\mathcal{P}_{noRW}$. Obviously, if the hosting page uses a JavaScript library that defines many dangerous functions, the untrusted guest code would have to be restricted to prevent access to these functions. It appears natural to treat the ADsafe problem more conservatively than FBJS, since FBJS code is executed in a browser first augmented with the defenses provided by Facebook, whereas ADsafe code is executed in a hosting page (provided by an arbitrary publisher) that may contain other scripts that inadvertently circumvent the sandboxing provided by the ADsafe libraries.

6. Conclusions

We have studied methods for filtering and rewriting untrusted code, using Yahoo! ADsafe and Facebook FBJS as illustrative and motivating examples. Using sublanguages Jt and Js, we show how to filter untrusted JavaScript to prevent access to any property names not manifest in the code, or to prevent access to scope objects, including the global scope object. Further, provable properties of sublanguage Jg show that access to the global object can be achieved by the kind of semantic restrictions imposed by wrapper functions such as $FBJS.ref. We also prove that subset Jt supports variable renaming, which is not semantic-preserving for JavaScript code outside Jt. A corollary is that renaming of global properties of Jt code isolates Facebook applications from each other, effectively providing separate namespaces. We also prove that renaming can be used to prevent interaction between untrusted code and blacklisted objects and properties, such as might be defined by a page hosting untrusted content.

Applications for Secure JavaScript Subsets. The subsets we have defined are very close to our two main reference real-world subsets FBJS and ADsafe, as described in Section 5. Another real-world field of application for our subsets is that of safe JavaScript widgets. A recent study by Livshits and Guarnieri [13] analyzes 8379 real-world widgets used on the Microsoft Vista Sidebar, Microsoft Windows Live and iGoogle and shows that the percentage of widgets that use the features forbidden in our subsets are very small. For example, eval is used only in 0.4 percent of Live gadgets, and 4.7 percent of Google gadgets. Member access is used only in 6.5 percent of Live gadgets, and 16.4 percent of Google gadgets. These figures do not take into account the cases where a widget could be re-written to avoid using the offending construct. The main concern of a widget host is once again to isolate widgets from the surrounding environment, and can be addressed by the properties of our subsets.

Future Work. An alternative to code rewriting that we have not examined in detail is to simply delete or redefine potentially harmful properties, such as property valueOf of Object.prototype and properties sort, reverse and concat of Array.prototype. This could allow additional code to be executed harmlessly. However, the effectiveness of this method requires further investigation because different browsers treat deletion of native objects differently. For example, deleting properties works in Safari, because deletion is permanent, but does not work in Firefox, for example, because executing delete Array; reinstates Array, Array.prototype and its original property sort, and similarly for the other cases.

In this paper we focussed on identifying appropriate syntactic restrictions to define secure subsets of JavaScript. In ongoing work [18] we are developing techniques to analyze new and existing run-time checks that can be inserted automatically in the code in order to design larger subsets. In particular, we would like to analyze the security properties of wrapper functions, and restricted forms of eval and member access e[e].

Proving formal properties for a practical programming language as extensive as JavaScript, without the help of an automatic tool, has been possible, but taxing. In future work, we plan to improve the usability of our framework by extending the coverage of our semantics to browser-specific cases and developing a tool to partially-automate the proofs. Indeed, many other scenarios involving the cooperation of trusted and untrusted JavaScript code lend themselves naturally to be studied following our approach.

Acknowledgments. We thank Douglas Crockford of Yahoo! and Ryan McGeehan of Facebook for their helpful comments and enthusiasm. We thank Adam Barth and the reviewers of CSF for useful comments and suggestions. We are indebted to John Mitchell for his invaluable contribution to the research leading to this paper. Sergio Maffeis is supported by EPSRC grant EP/E044956 /1. Ankur Taly acknowledges the support of the National Science Foundation.

Appendix

1. Well-formedness

We give here precise definitions for the informal notions of well-formedness mentioned in Section 3 and Section 4.

Definition 10: A state $S = (H, l, t)$ is *well-formed*, denoted by $Wf(S)$, if and only if its heap, scope and term are well-formed, denoted respectively by $Wf_{\mathcal{H}}(H)$, $Wf_{\mathcal{S}}(l)$ and $Wf_{\mathcal{T}}(t)$.

- A term t is well-formed iff it can be derived using the grammar rules consisting of both the language constructs and the internal constructs, and all heap addresses contained in t are allocated ie $l \in t \Rightarrow l \in dom(H)$.
- A scope address $l \in dom(H)$ is well-formed iff the scope chain starting from l does not contain cycles, and $(\text{@Scope} \in prop(H(l))) \wedge (H(l).\text{@Scope} \neq null \Rightarrow Wf_{\mathcal{S}}(H(l).\text{@Scope})$.
- A heap H is well-formed iff the following conditions are true
  - Every object in the heap must have @Class and @Prototype in its set of properties.
  - Every function object in the heap must have @Call, @Scope, length, @Body and @Prototype in its set of properties.
  - Every arguments object in the heap must have callee and length in its set of properties.
  - Every array object in the heap must have the length property. The length property must always contain a number.
  - Every native function object must have the @Actuals property.
  - Every String, Number and Boolean object must have the @Value property.
  - All native error objects must have the message property.
  - #Global (that is, l_G) must be an allocated address and must at least have the this property.
  - The prototype chain for any object must never contain a cycle.
  - The scope property for any function object must contain a well-formed scope address and also the body property must contain a well-formed term.

Given a subset of JavaScript user terms 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^*.

2. Renaming of States

We give here the complete definition of renaming for states.

Definition 11: Let $S = (H, l, t)$ be a state and α be a safe renaming for S. We define $\alpha(S)$ as the state $(\alpha(H), l, \alpha(t))$ where:

- $\alpha(H)$ is defined as H where
  - all functions of the form fun([x˜]){P} stored in some object in H are renamed to fun([α(x˜)]){α(P)} (α is applied to x only if $x \in dom(\alpha)$);
  - for all addresses l such that @Scope $\in H(l)$, every property name mp $\in H(l)$ such that Prop2Id$(mp) \in dom(\alpha)$ is renamed to Id2Prop(Prop2Id(α(mp))) (without changing the property attributes);
- $\alpha(t)$ is defined as t where:
  - every identifier x in t such that $x \in dom(\alpha)$ is renamed to α(x);
  - every property name mp in t such that Prop2Id$(mp) \in dom(\alpha)$ is renamed to Id2Prop(Prop2Id(α(mp))) (without changing the property attributes) iff it appears in one of the following contexts:
    1) l$*-$ where @Scope $\in H(l)$ is true OR $l =$null
    2) l.@Put($-$,va) where @Scope $\in H(l)$ is true OR $l =$null
    3) function $-$(x˜){P}
    4) try (Throw,va,xe) catch ($-$) {s1} [finally {s2}].

References

[1] I. Aktug, M. Dam, and D. Gurov, "Provably correct runtime monitoring," in *Proc. of FM 2008*, ser. LNCS, vol. 5014. Springer, 2008, pp. 262–277.

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

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

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

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

[6] ——, "ADsafe: Making JavaScript safe for advertising (2007 version)," http://web.archive.org/web/20071225101246/http://www.adsafe.org/, 2007.

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

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

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

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

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

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

[13] B. Livshits and S. Guarnieri, "Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code," MSR-TR-2009-16, Feb. 2009.

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

[15] ——, "An operational semantics for JavaScript," in *Proc. of APLAS'08*, ser. LNCS, vol. 5356, 2008, pp. 307–325, See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[16] ——, "An operational semantics for JavaScript," in *Proc. of APLAS'08*, ser. LNCS, vol. 5356. Springer Verlag, Dec. 2008, pp. 307–325.

[17] ——, "Language-based isolation of untrusted JavaScript," Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[18] ——, "Run-time enforcement of secure javascript subsets," in *Proc of W2SP'09*. IEEE, 2009.

[19] Prototype Core Team, "Prototype JavaScript framework: Easy Ajax and DOM manipulation for dynamic web applications," http://www.prototypejs.org.

[20] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of Dynamic HTML," *ACM Transactions on the Web*, vol. 1, no. 3, 2007.

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

[22] The FaceBook Team, "FaceBook," http://www.facebook.com/.

[23] ——, "FBJS," http://wiki.developers.facebook.com/index.php/FBJS.

[24] ——, "FBML," http://wiki.developers.facebook.com/index.php/FBML.

[25] P. Thiemann, "Towards a type system for analyzing javascript programs," in *Proc. of ESOP'05*, ser. LNCS, vol. 3444, 2005, pp. 408–422.

[26] ——, "A type safe DOM API," in *Proc. of DBPL'05*, 2005, pp. 169–183.

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

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