Language-Based Isolation of Untrusted JavaScript

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

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

Ankur Taly
Department of Computer
Science, 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 study 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 shortcomings, and give a foundation for improved solutions based on an operational semantics of the full ECMA262-3 language. We also discuss how to apply our analysis to address the 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 [5], 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 serve untrusted content, such as applications developed by users, without isolating this content in an iframe. Instead, a number of current sites pre-process untrusted content, applying filters or source-to-source rewriting before the content is served.

In this paper, we study filtering and rewriting methods for managing untrusted JavaScript [6, 9], focusing on two illustrative examples: Yahoo! ADsafe and Facebook FBJS. Facebook [7] is a leading social networking site that makes substantial use of JavaScript, allowing user-originated code to interact with trusted libraries. Yahoo's ADsafe [1] proposes a particularly flexible 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 also uses JavaScript rewriting to run applications in a "separate namespace" and insert certain runtime checks. While Google Caja [3] 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) [10], available on the web [11] and described previously in [12, 13]. We initially tried to use this framework to prove isolation properties of ADsafe and FBJS, but failed because of problems we discovered. As explained in Section 2, ADsafe 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. 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 with 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 make it possible to identify the 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 the global object. Because of the size of the operational semantics for the full ECMA-262 language [10], approximately 60 pages of ascii text, each of these proofs reflects significant effort.

Related work on language-based methods for isolating the effects of potentially malicious web content include [15], which examines ways to inspect and cleanse dynamic HTML content, and [20], which modifies questionable JavaScript, for a more restricted fragment of JavaScript than we consider here. A short workshop paper [19] also gives an architecture for server-side code analysis and instrumentation, without exploring details or specific methods for constraining JavaScript. Foundational studies of much more limited subsets of JavaScript are reported in [4, 18, 20]; see [12].

Plan of the paper. In Section 2, we describe FBJS, ADsafe, and vulnerabilities we discovered. Language properties supporting filtering and rewriting are discussed in Section 3. In Section 4, we briefly review our previous work [12] on JavaScript operational semantics. In Section 5 we use the operational semantics to identify safe subsets of JavaScript, and prove their properties. In Section 6, we discuss how our results can solve the problems found in FBJS and ADsafe, and discuss the solutions currently adopted. Concluding remarks are in Section 7.

2. JavaScript ISOLATION EXAMPLES

In this Section, we summarize the Facebook and ADsafe isolation mechanisms and explain some of the problems we observed with them. The $FBJS_{08}$ vulnerability we describe was reported to Facebook and has been repaired. Similarly, the deficiency we observed in $AdSafe_{07}$ 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 [7] 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 (among 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 provide a richer user experience and affect the way a user profile is displayed.

Facebook applications are written in FBML [17], 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 [16] 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 at the same time 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, separating 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 allow FBJS code to contain the special identifier `this`. 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 (see Section 6 for further discussion of `ref` and the revised version `$FBJS.ref` now used). Without this wrapper, code such as `(function (){return this})()` could return the `window` object, which would give the application access to the full Facebook page. A similar problem arises with the notation `object["property"]`, which is similarly rewritten to `a12345_object[idx("property")]`. Functions `ref` and `idx` are designed to prevent `this` from being bound to the `window` object and prevent `"property"` from being a blacklisted property. Other syntactic checks on FBJS code exclude the use of dangerous constructs such as `eval` and `with`, and prevent use of special properties of objects such as `__parent__`, `constructor`, `valueOf`, and so on, which may be used in some web browsers to access (indirectly) the `window` object.

Two Facebook vulnerabilities found

We initially attempted to use our operational semantics of JavaScript [11] 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 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 early October, 2008 as $FBJS_{08}$.

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)` refers to the current value of `this`, which may be the `window` object if this is the current scope. This access to the `window` object allows a $FBJS_{08}$-application-based attacker to take over the page; see Felt *et al.* [8] for discussion of further ramifications.

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

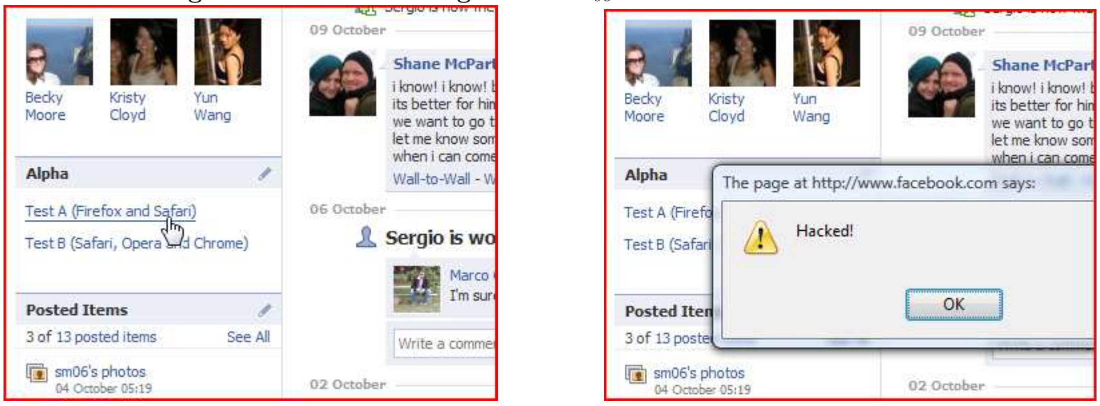
```
try {throw (function(){return this});} catch (get_scope){...}}
```

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.

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



Recurring subtleties. There is another, even more subtle way to access the scope object, by the code

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

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 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 the this identifier is once again bound to the scope object, which escapes the ref check, and can be returned by the recursive call. Additional code can disable ref as described above and escape from the sandbox.

If we invoke get_window(1) then in the else branch we can enable the attack described above by using the expression get_scope(0) to return the current scope object, because once again the this identifier of the recursive function is bound to its own scope object, which escapes the ref check.

Demonstration

JavaScript for the $FBJS_{08}$ attacks described above appears in Figure 2; screen shots are in Figure 1. While the attacks illustrate web site vulnerabilities, their effectiveness is browser-dependent because of deviations from the ECMA262-3 specification. Because 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 the this identifier. 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 [5].

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 [1], "*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 JSLintso 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.

Using our JavaScript operational semantics [11, 12], we tried to prove that the 2007 version of ADsafe [2] indeed isolated ADsafe-conformant JavaScript code from the global object.

In setting up the proof, however, we found a problem with the ADsafe design: the page hosting a ADsafe-conformant advertisement may unwittingly define objects or add properties to accessible objects in a way that provides access to the global object. For example, ADsafe-validated code cannot use the identifier this because, for example, a function such as function f(){return this} returns the global object. If the page hosting an advertisement adds function f to Object.prototype, then the ADsafe-compliant code var o={};o.f() evaluates to the global object (because o inherits from Object.prototype), violating the intended confinement property.

In fact, we found that a very common JavaScript library, *prototype.js* [14], provides ways for ADsafe-compliant code to access the global object. 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 6). However, these restrictions are not specified with the same precision as other ADsafe guidelines, leading us to believe that further investigation is warranted.

Figure 2: *FBJS*$_{08}$ **exploit code.**

```
<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>
```

3. FILTERING AND REWRITING

We have seen two basic language-based methods for protecting sites from untrusted JavaScript in Section 2. One method, illustrated by ADsafe, involves checking whether the untrusted content belongs to a subset of JavaScript that does not access portions of the page. Alternatively, as illustrated by Facebook rendering of FBJS, source JavaScript may be rewritten to constrain or isolate its effects. A very natural JavaScript rewriting method renames variables, so that the untrusted content is effectively run in a "separate namespace" from other portions of the page. Another rewriting method, also used for FBJS, involves wrapping some constructs, such as this, with another function that restricts the possible values of this. While these methods may seem largely straightforward, there are a number of subtleties related to the expressiveness and complexity of JavaScript. Further, since organizations that have devoted significant time and effort to deploying language-based methods have overlooked certain problems, we believe that a systematic study based on traditional programming language foundations will be useful for improving the security of future sites.

One basic issue is that in JavaScript, every heap access by the program involves a property p of an object o that may be a scope object containing the local variables of some block (including the global scope), or a non-scope object created and manipulated directly by the program. The problem of determining which object-property accesses o.p are made by a program can be divided into determining which objects o are accessible, and which properties p are named and used.

Property names. The first technical problem we consider is to determine a finite set of properties that may be accessed by given code. This 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, the problem becomes manageable if we eliminate operations that may convert strings to property names, such as eval and `o[ ]`.

We must also consider implicit access to native properties that may not be mentioned in the code at all. 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, returning the string "an␣[object␣Object]". (If o does not have the toString property, then it is inherited from its prototype). Fortunately, all property names accessed implicitly come from a finite set of native property names [13].

Definition 1. *Let* $Prop_{nat}$ =

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

be the set of property names that are involved in implicit properties accesses.

Note that all these involve *getting* the value of the property and not *setting* it.

Thus the first problem we address is:

Problem 1. *Define a meaningful sublanguage* $J_{textual}$ *and prove that for any program* $P \in J_{textual}$*, if execution of* P *accesses property* p *of some object, then either* $p \in Prop_{nat}$ *or* p *appears textually in* P*.*

Isolating the Global object. Untrusted JavaScript should be restricted from accessing the global object, since untrusted code can access data belonging to trusted code, or untrusted code associated with a different application, if it has access to the global object. Thus the second problem we address is:

Problem 2. *Find a meaningful sublanguage* J_{global} *and prove that no* J_{global} *code can access the global object.*

As explained in Section 5, our proposed J_{global} is a subset of $J_{textual}$, because the constructs that convert strings to properties can otherwise be used to reach the global object.

We provide two solutions to Problem 2, one based on a syntactic subset defined by a subset of the JavaScript grammar. For reasons explained in Section 2 in connection with the vulnerability we discovered in *FBJS*$_{08}$, this syntactic subset cannot contain the keyword this. However, the reason why this is problematic is that it *sometimes* refers to a scope object or the global object. By semantically restricting the value of this, as may be achieved by a correct wrapper ref as in Section 2, we can allow this in a semantically restricted and syntactically larger sublanguage J^s_{global}.

Identifer renaming. The final technical problem we consider is the ability to rename identifiers in JavaScript code. Syntactic renaming is difficult for full JavaScript, because property names (and therefore variable names, if an object variable o may refer to a scope object) may be computed by string operations. However, we prove that the subsets J_{global} and J^s_{global} allow variable renaming.

Problem 3. *Identify meaningful sublanguages of JavaScript that support semantics-preserving capture-avoiding renaming of identifiers.*

We can also use renaming and solutions to previous problems to keep untrusted code from accessing certain critical properties:

Figure 3: Metavariables and Syntax for Values.

```
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
```

PROBLEM 4. *Given a list of properties $\mathcal{P}_{noW}$ and $\mathcal{P}_{noRW}$, define a subset $J_{\mathcal{P}_{noW},\mathcal{P}_{noRW}}$ such that any program in the subset cannot write to any properties in $\mathcal{P}_{noW}$ and can neither read nor write to any properties in $\mathcal{P}_{noRW}$.*

4. JAVASCRIPT SEMANTICS

In this Section we briefly summarize our formalization of the operational semantics of JavaScript [11, 12, 13] based on the ECMA262-3 standard [10], and introduce some auxiliary notation and definitions. In [12], 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 considered in the section 5. Further discussion of the relation between this semantics and current browsers implementations appears in [12].

4.1 Operational Semantics

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.

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. We denote by *Expr*, *Stmnt* and *Prog* the sets of all legal JavaScript expressions, statements and programs. We define $\texttt{Terms}_{js}$ = *Expr* ∪ *Stmnt* ∪ *Prog* as the set of all JavaScript terms. For each class of term we 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. We call such a triple (H, l, t) a *state*, denoted by S. Conversely, we use the notation *heap*(S), *scope*(S) and $\texttt{term}(S)$ to denote each component of the state. Predicate $Wf(S)$ denotes that state S is well-formed (see [13] for a definition of well-formedness). A state S is *initial* if $Wf(S)$ and $\texttt{term}(S)$ is a user term.

DEFINITION 2. *(Reduction Trace) The* reduction trace *of a state S is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$.*

By the progress property shown in [13], we know that for every well-formed state S there is only one possible trace. Given a trace τ, we denote by $First(\tau)$ its first state. If the trace is finite, we denote by $Final(\tau)$ its final state. Again by the progress property, the evaluation of expressions returns either a value or an exception, and the evaluation of statements and programs terminates with a completion (explained below).

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 bigger term including the former as a sub-term. In general, 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 $t < S$ or $t != t'$ or $f(a) = b$ for set membership, inequality and function application.

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(l',P) (an internal expression used to evaluate the body of a function) reduces in one step to H1,l1,@FunExe(l',P1). The rule below show exactly 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 [11] 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 5 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 statement.

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.

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

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

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 straightforward 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.

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).

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

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

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

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. For example, the global object defines properties to store special values such as &NaN and &undefined, functions such as eval, toString 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.

The object @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).

5. FORMAL ANALYSIS

In this Section, we present a formal analysis of the problems discussed in Section 3. Using only syntactic restrictions we propose a subset $J_{textual}$ which solves Problem 1 – the names of all properties not in $Prop_{nat}$ accessed during the execution of a program must appear textually in the program – and a smaller subset J_{global} which solves Problems 2 and 3 – no term returns the global object, and the meaning of a well-formed term does not change after capture-avoiding renaming of identifiers. Using a semantic restriction amounting to run-time checking, we also present a subset J^{s}_{global}, less restrictive than J_{global}, which enjoys the same properties.

Definitions. We start adding some more notations and definitions to the operational semantics described in Section 4. A *meta-call* $(f, \langle args \rangle)$ is a pair where f is a function or predicate appearing in the premise of a reduction rule, and $\langle args \rangle$ is the list of its actual arguments as instantiated by a reduction step according to such a rule. For every state S, we denote by $Fcall_1(S)$ 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 $Fcall(S)$ 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)$\}$, where mp denotes the name of the property being accessed (using a prefix notation for the functions defined in Section 4).

Definition 3. *(Single step property access) For any state S, we define $Pacc(S)$ as the set of all property names accessed during a single step state transition. Formally,*

$$Pacc(S) = \{mp \mid \exists f \in \mathcal{F}_H \; \exists H, l : (f, \langle H, l, mp \rangle) \in Fcall(S)\}$$

For any trace $Tr(S)$, we denote by $Pacc(Tr(S))$ the set $Pacc(S) \cup \bigcup_{Si \in Tr(S)} Pacc(S_i)$.

A subset J of JavaScript is a subset of the terms $\mathtt{Terms}_{js}$ that are derivable from the internal and user grammars. (Recall that the internal grammar provides symbols only used to define the operational semantics, not for writing user programs.) Any subset J can be partitioned in the set of all user and internal terms of the subset: $J = J^{user} \uplus J^{int}$.

For a valid user-defined (non-native) program code P and a heap H, we say that $P \in H$ iff P is contained in some property of some object on the heap.

Definition 4. *(Wellformedness) For any subset J, we define a wellformedness predicate on states $Wf_J(S)$ by*

$$Wf_J(H, l, t) \;=\; Wf_{Heap}(H) \wedge Wf_{scope}(l) \;\wedge\; t \in J \\ \wedge\; \forall P \in H \;\; P \in J^{user}$$

Definition 5. *A JavaScript subset J enjoys the* preservation property *iff $\forall S : Wf_J(S) \wedge S \to S' \Rightarrow Wf_J(S')$.*

In [12, 13], we showed progress and preservation for the whole of JavaScript. Since any subset corresponds to a subset of JavaScript terms, progress holds also for any subset J. However, preservation may not hold for a subset, since there could be a term in the subset that reduces to one outside of the subset. We will refer to a subset enjoying preservation as a *closed subset.* Closed subsets are semantically meaningful sublanguages.

5.1 Isolating property names: $J_{textual}$

The first subset we define is $J_{textual}$. It guarantees that all the properties (not in $Prop_{nat}$) accessed during execution appear textually in the code. As illustrated in Section 3, we need to avoid certain constructs that make it possible to access properties whose names are not present textually in the term. As a technical mechanism, we separate out property names, strings and identifiers in the operational semantics and denote them respectively by mp, m and x. We make all the implicit conversions between them explicit by adding meta conversion functions: Id2Prop, Str2Prop, Prop2Str. The semantics already contained explicit conversion of strings to programs: ParseProg, ParseFunction, ParseParams.

Defining $J_{textual}$. Our first concern is to exclude all terms whose reductions involves a call to either one of ParseParams, ParseFunction, ParseProg or Str2Prop. Then, we need to exclude all terms which reduce in one step to a term outside the subset. For example the constructor property of a function points to the Function object, which calls ParseFunction when it is invoked as a constructor. Hence, constructor must be excluded too. After analyzing the semantics in detail we have reached a definition for $J_{textual}$.

Definition 6. *The subset $J_{textual}$ is defined as $\mathtt{Terms}_{js}$ 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; all internal terms reachable during the reduction of the above terms.*

Note that constructor $\in Prop_{nat}$, so the constructor property may be accessed implicitly in some cases. However, we prove (Theorem 1) that all the implicit accesses are safe and do not return the Function object.

Formal Analysis. We begin by defining formally the property $P_{textual}$ which implies that the names of all properties not in $Prop_{nat}$ accessed during the execution of a program must appear textually in the program. Let $IN(S) = \{x | x \in \mathtt{term}(S)\}$ and $PN(S) = \{mp | mp \in \mathtt{term}(S)\}$. We define $HN(S)$ as the sets of identifiers present in all program code stored on the heap (example, code from other function declarations). Therefore, $HN(S) = \{x | \; x \in P \text{ and} P \in heap(S)\}$. Moreover, $N(S) = IN(S) \cup HN(S)$.

Definition 7. *($P_{textual}$) For any trace τ, the predicate $P_{textual}(\tau)$ holds iff:*

$$Pacc(\tau) \subseteq \mathit{Id2Prop}(IN(First(\tau))) \;\cup\; Prop_{nat}.$$

We now show that $P_{textual}$ holds for all traces that start from a well-formed state. Consider a reduction step from S to S'. First, we show that all property names accessed during the reduction step are either present in S or in $Prop_{nat}$. Then, we show that any property name that appears in S', is present either in S or in $Prop_{nat}$.

Theorem 1. *$J_{textual}$ is closed under reduction.*

Lemma 1. *For all well-formed states S in $J_{textual}$*

$$Pacc(S) \subseteq PN(\mathtt{term}(S)) \;\cup\; Prop_{nat}.$$

Lemma 2. *For all well-formed states S_1, S_2 in $J_{textual}$, if $S_1 \to S_2$ then $IN(S_2) \subseteq IN(S_1)$, $HN(S_2) \subseteq N(S_1)$,*

$$PN(S_2) \subseteq PN(S_1) \cup \mathit{Id2Prop}(IN(S_1)) \cup Prop_{nat}.$$

We assume that the heap for the initial state S_{init} contains only the native objects. Therefore $HN(S_{init}) = \emptyset$.

THEOREM 2. *For all well-formed initial states S in the subset $J_{textual}$, $Pacc(Tr(S)) \subseteq$ Id2Prop$(IN(S)) \cup Prop_{nat}$.*

Theorem 2, which follows from Lemma 1 and Lemma 2 above essentially says that the reduction traces of all well-formed initial states have the desired property $P_{textual}$. As a result, $J_{textual}$ is a valid solution for Problem 1.

5.2 Isolating the global object: J_{global}

We now define the subset J_{global} which guarantees that no term can return the global object. We assume that the heap of the initial state defines only the global object and the standard native objects. According to the semantics, the global object is only accessible via the internal properties @scope and @this. These internal properties can only be accessed as a side effect of the execution of other instructions. In particular, 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 final result of a reduction step. Hence, the @this property is the only way for a term to return the global object. The simplest way is via the this identifier. Its semantic rule is:

$$\frac{\text{Scope(H,l,@this)=l1} \qquad \text{H,l1.@Get(@this)=va}}{\text{H,l,this} \longrightarrow \text{H,l,va}}$$

When the scope l is the global object, so is va.

Besides using this, the global object can be returned by extracting and calling in the global scope the functions concat, sort or reverse of Array.prototype, and valueOf of Object.prototype. For example, var f=Object.prototype.valueOf;f() evaluates to the global object. Since it is difficult to statically determine the type of the base object whenever a property is accessed, we conservatively forbid accessing these properties altogether. In order to do so, we define J_{global} as a subset of $J_{textual}$, so that we can force property names to appear textually in the code.

Defining J_{global}. After analyzing the semantics in detail, we have settled on the following definition for J_{global}.

DEFINITION 8. *The subset J_{global} is defined as $J_{textual}$, minus: all terms containing the expression this; all terms containing the identifiers valueOf, sort, concat and reverse; all internal terms reachable during the evaluation of the above terms.*

Note that since valueOf $\in Prop_{nat}$, it may be accessed implicitly for some objects. However, we prove (Theorem 3) that all the implicit accesses are safe and do not return the global object.

Formal Analysis. Let $Value$ be a function that given a state returns a non-null value only if the state corresponds to a final value or an exception, that is $Value(S) = vae$ if $\texttt{term}(S) = vae$ or $\texttt{term}(S) = (ct, vae, xe)$, and $Value(S) = null$ otherwise. We can formally define a property P_{global} which implies isolation of the global object.

DEFINITION 9. *(P_{global}) For any trace τ, $P_{global}(\tau)$ holds iff $Value(Final(\tau)) \neq l_{global}$.*

We now show that P_{global} holds for all reduction traces starting from well-formed states. First, we define a *goodness* property of heaps and show that whenever $good(H)$ holds (where H denotes the current heap), the single step reduction of a term never returns the global object. Then, we show that for every single step reduction, if the initial heap is good, then the final heap is good too.

DEFINITION 10. *(Heap goodness) We say that a heap is* good*, denoted by $Good(H)$, iff H is such that the global object is only reachable via the @scope and @this properties.*

$$Good(H) \Leftrightarrow \forall l, p : H(l).p = l_{global} \quad \Rightarrow \quad \begin{array}{l} p = \texttt{@scope} \vee \\ p = \texttt{@this} \end{array}$$

Let H_{init} be the initial heap. Since the global object is only reachable via the @scope or @this properties, H_{init} is good.

THEOREM 3. *J_{global} is closed under reduction.*

LEMMA 3. *For all well-formed states S (in the subset J_{global}),*

$$Good(heap(S)) \wedge S \rightarrow S' \Rightarrow Value(S') \neq l_{global}$$

LEMMA 4. *For all wellformed states S and S',*

$$Good(heap(S)) \wedge S \rightarrow S' \Rightarrow Good(heap(S'))$$

THEOREM 4. *For all wellformed initial states S in the subset J_{global}, $Value(Tr(S)) \neq l_{global}$.*

Theorem 4 says that the reduction traces of all well-formed initial states enjoy P_{global}, therefore J_{global} is a valid solution for Problem 2.

5.3 A semantic subset: J^s_{global}

In the subset J_{global}, we disallow the this identifier because its reduction could potentially return the global object. However, there are several cases when the @this property of the current scope object does not contain the global object and therefore can be used safely. For example in

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

the this identifier is bound to object o during the execution of o.f(). Disallowing the identifier this completely may be deemed too restrictive for certain purposes. For example, many existing JavaScript libraries would need extensive rewriting. Therefore, we investigate a subset containing the identifier this, with guarantees that this cannot be used to access the global object. For concreteness, we impose a restriction on the semantics of this so that it returns a harmless value (say null) if the @this property of the current activation object contains the global object, and its proper value otherwise.[1] Thus, we consider an alternative operational semantics for the this expression:

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

In the modified semantics, we can define the subset J^s_{global} as J_{global} plus all terms containing this. The theorems that we stated for J_{global} hold for J^s_{global} in the modified semantics.

THEOREM 5. *J^s_{global} is closed under reduction.*

THEOREM 6. *For all well-formed initial states S in the subset J^s_{global}, $Value(Tr(S)) \neq l_{global}$.*

[1] In practice, the semantic restriction can be implemented (as done by FBJS) by rewriting user code so that every occurrence of this becomes ref(this), where ref behaves as described above. As shown by the attacks reported in Section 2 though one need also to make sure that the function ref cannot be redefined by user code.

5.4 Closure under renaming

In Section 3 we discussed how renaming all the identifiers of the code is used to effectively execute an application in a separate namespace. We now show that the subsets J_{global} and J^s_{global} are closed under capture-avoiding renaming of identifiers. A corollary of Theorem 2 is that this renaming has a semantically isolating effect.

DEFINITION 11. *Given a set of identifiers $\mathcal{S}$, a partial function α from identifiers to identifiers is a* capture avoiding (CA) renaming compatible with $\mathcal{S}$ *iff, for all x in the domain of α, $\alpha(x) \notin \mathcal{S}$.*

We extend renaming to terms, heaps and states and traces in the obvious way. For any term t, $\alpha(t)$ denotes a new term where all the identifiers x occurring in t have been renamed to $\alpha(x)$. Note that while renaming terms we only rename identifiers and not property names. Therefore o.p gets renamed to a12345_o.p and not a12345_o.a12345_p. For any well-formed heap H, $\alpha(H)$ denotes a heap where the property names for all the *activation objects* (any object with the internal property IsActivation) and the identifiers of each $P \in H$ have been renamed according by α. Finally, $\alpha(H, l, t) = (\alpha(H), l, \alpha(t))$ and $\alpha(S_1, \ldots, S_n, \ldots) = \alpha(S_1), \ldots, \alpha(S_n), \ldots$. In general, we will require a CA renaming to be compatible with all the identifiers of the state to which is applied, plus the identifiers in $Prop_{nat}$, so to effectively avoid accidental capture of names.

We prove that the intended meaning of a program does not change under renaming by showing that renaming is preserved under reduction.

DEFINITION 12. *Given two states S_1, S_2 and an arbitrary function α, we define the relation $\sim_\alpha$ by $S_1 \sim_\alpha S_2$ iff $S_2 = \alpha(S_1)$.*

DEFINITION 13. *(P_{ren}) The renaming property P_{ren} states that for all well-formed initial states S, if α is a CA renaming compatible with $N(S) \cup Prop_{nat}$, then $\alpha(Tr(S))$ equals $Tr(\alpha(S))$.*

The main result of this section is that $\sim_\alpha$ is preserved under reduction for both subsets J_{global} and J^s_{global}. A direct corollary of this result is that these subsets enjoy the renaming property P_{ren} and are therefore solutions for Problem 3 described in Section 3.

THEOREM 7. *For all well-formed states S_1 and S_2 both in either J_{global} or J^s_{global}, and all CA renamings α compatible with $N(S_1) \cup N(S_2) \cup Prop_{nat}$,*

$$S_1 \sim_\alpha S_2 \ \wedge \ S_1 \rightarrow S_1' \Rightarrow \ S_2 \rightarrow S_2' \ \wedge \ S_1' \sim_\alpha S_2'.$$

5.5 Corollary : restricting access

We now use the subsets that we have defined so far to find a solution to the final problem of restricting the untrusted code to only have readonly access to set of properties ($\mathcal{P}_{noW}$) and no access at all to another set $\mathcal{P}_{noRW}$. The solution $J_{\mathcal{P}_{noW}, \mathcal{P}_{noRW}}$ that we propose currently works only if we have the additional condition that $Prop_{nat} \cap \mathcal{P}_{noRW} = \emptyset$. Also it is more conservative and disallows any both read and write access to all properties in the set $\mathcal{P}_{noW} \cup \mathcal{P}_{noRW}$, except those in $Prop_{nat}$, which in any case have only implicit read access. We formally define the subset $J_{\mathcal{P}_{noW}, \mathcal{P}_{noRW}}$ as the subset J_{global} minus: all terms containing an identifier from the set $\mathcal{P}_{noW} \cup \mathcal{P}_{noRW}$. From theorem 2 and lemma 3, we obtain the following corollaries.

COROLLARY 1. *$J_{\mathcal{P}_{noW}, \mathcal{P}_{noRW}}$ is closed under reduction.*

COROLLARY 2. *For all well-formed initial states S in the subset $J_{\mathcal{P}_{noW}, \mathcal{P}_{noRW}}$, $Pacc(Tr(S)) \ \cap \ ((\mathcal{P}_{noRW} \cup \mathcal{P}_{noRW}) - Prop_{nat}) = \emptyset$.*

In the next section, we use the subset $J_{\mathcal{P}_{noW}, \mathcal{P}_{noRW}}$ to solve a simplified version of the ADsafe problem.

6. APPLICATIONS: FBJS AND ADsafe

In this Section we discuss how FBJS and ADsafe have changed to address the problems we explained in Section 2, and we compare the currently-adopted solutions with the JavaScript subsets proposed in Section 5.

6.1 Fixing FBJS

Within hours of our disclosure to them, the Facebook team addressed the problems discussed in Section 2 by separating the namespace of the run-time checks ref and idx from the namespace available to FBJS applications. They added the functions implementing the run-time checks as properties of a private object $FBJS, and statically prevent user code from including the $ character in an identifier. This thwarts the attacks reported in Figure 2 because an expression like get_scope().$FBJS.ref is rewritten into the ill-formed code a12345_get_scope()..ref.

A similar solution justified by our analysis is to use the subset J^s_{global}, and perform a simple syntactic check that $FBJS does not syntactically appear in user code. By Theorem 2, $FBJS cannot be tampered with. By Theorem 6, the global object cannot be accessed, and by Theorem 7 the FBJS renaming discipline preserves the meaning of programs. The alternative semantics of the this identifier is faithfully implemented by the replacing each occurrence of this in the code by the expression (this===window?null:this). Hence, our theorems justify the approach behind the security mechanism of FBJS, although currently FBJS uses more extensive rewriting and fewer syntactic restrictions.

We also note that J_{global} provides a syntactically checkable alternative that could be useful in some Web applications.

6.2 Enforcing ADsafe

Shortly after we notified Yahoo! of the problems described in Section 2, the ADsafe [1] 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 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.

Our analysis supports a slightly different approach, in which we assume some aspects of the hosting page are known, and filter ads that might access forbidden properties of the page. More specifically, suppose that by statically analyzing the hosting page libraries we can extract sets of property names $\mathcal{P}_{noW}$ and $\mathcal{P}_{noRW}$, such that all "malicious damage" only arises from writing to a property in the set $\mathcal{P}_{noW}$ or either 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 $Prop_{nat}$. The set $\mathcal{P}_{noRW}$ could include security-critical properties such as eval, window, cookie, and other properties and methods that can be invoked to reach these. We have not developed an analysis method for examining libraries such as prototype.js, but it may be possible to do so using the call-graph of the native functions.

Given these sets we can define the subset $J_{\mathcal{P}_{noW},\mathcal{P}_{noRW}}$ for the untrusted code. Soundness of this approach essentially follows from Corollaries 1 and 2. The level of syntactic restriction for the subset $J_{\mathcal{P}_{noW},\mathcal{P}_{noRW}}$ would depend on how large or small the sets $\mathcal{P}_{noW}$ and $\mathcal{P}_{noRW}$ are. Thus, if the hosting page contains many dangerous functions, then the untrusted guest code that it could safely allow would be severely restricted.

It appears natural to treat the ADsafe problem more conservatively than FBJS, in the following sense. In FBJS, the browser is first augmented with the defenses provided by Facebook libraries and then exposed to sanitized untrusted code, whereas in ADsafe, the hosting page is provided by an arbitrary publisher who may be subjected to cross-site scripting or other web attacks that might contaminate the ADsafe libraries.

7. CONCLUSIONS

We have studied methods for filtering and rewriting untrusted code, using Yahoo! ADsafe and Facebook FBJS as illustrative and motivating examples. Using sublanguages $J_{textual}$ and J_{global}, we show how to filter untrusted JavaScript to prevent access to given properties or the global object. Further, provable properties of sublanguage J^s_{global} 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 these subsets support renaming, which is not semantic-preserving for JavaScript code outside these sublanguages. A corollary is that renaming of global properties of $J_{textual}$ 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.

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 both Array, Array.prototype and its original property sort, and similarly for the other cases.

Proving formal properties for a practical programming language as extensive as JavaScript, without the help of an automatic tool, has been possible, but very 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. Sergio Maffeis is supported by EPSRC grant EP/E044956 /1. Mitchell and Taly acknowledge the support of the National Science Foundation.

8. REFERENCES

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

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

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

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

[5] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *17th USENIX Security Symposium*, 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, 2008. ACM.

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

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

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

[12] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08.*

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

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

[15] 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.

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

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

[18] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, page 408422, 2005.

[19] K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Web 2.0 Security & Privacy (W2SP)*, 2008.

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