GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code

Benjamin Livshits
Microsoft Research

Salvatore Guarnieri
University of Washington

MSR-TR-2009-16

Microsoft® Research

Abstract

The advent of Web 2.0 has lead to the proliferation of client-side code that is typically written in JavaScript. This code is often combined or mashed-up with other code and content from disparate, mutually untrusting parties, leading to undesirable security and reliability consequences.

This paper proposes GATEKEEPER, a mostly static approach for soundly enforcing security and reliability policies for JavaScript programs. GATEKEEPER is a highly extensible system with a rich, expressive policy language, allowing the hosting site administrator to formulate their policies as succinct Datalog queries. The primary application of GATEKEEPER is in reasoning about JavaScript widgets such as those hosted by widget portals Live.com and Google/IG. Widgets submitted to these sites can be either malicious or just buggy and poorly written, and the hosting site has the authority to reject the submission of widgets that do not meet the site's security policies. To show the practicality of our approach, we describe nine representative security and reliability policies. Statically checking these policies results in 1,341 verified warnings in 684 widgets, no false negatives, due to the soundness of our analysis, and false positives affecting only two widgets.

1 Introduction

JavaScript is increasingly becoming the lingua franca of the Web, used both for large monolithic applications and small *widgets* that are typically combined with other code from mutually untrusting parties. At the same time, many programming language purists consider JavaScript to be an atrocious language, forever spoiled by hard-to-analyze dynamic constructs such as `eval` and the lack of static typing. This perception has lead to a situation where code instrumentation and not static program analysis has been the weapon of choice when it comes to enforcing security policies of JavaScript code [16, 19, 22, 28].

As a recent report from Finjan Security shows, widget-based attacks are on the rise [13], making widget security an increasingly important problem to address. The report also describes well-publicised vulnerabilities in the Vista sidebar, Live.com, and Yahoo! widgets. The primary focus of this paper is on statically enforcing security and reliability policies for JavaScript code. These policies include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, redirect and cross-site scripting detection, preventing global namespace pollution, taint checking, etc. Soundly enforcing security policies is harder that one might think at first. For instance, if we want to ensure a widget cannot call `document.write` because this construct allows arbitrary code injection, we need to either analyze or disallow tricky constructs like `eval("document" + ".write('...')")`, or `var a = document['wri' + 'te']; a('...');` which use reflection or even `var a = document; var b = a.write; b.call(this, '...')`, which uses aliasing to confuse a potential enforcement tool. A naïve unsound analysis can easily miss these constructs. Given the availability of JavaScript obfuscators [15], a malicious widget may easily masquerade its intent. Even for this very simple policy, `grep` is far from an adequate solution.

JavaScript relies on heap-based allocation for the objects it creates. Because of the problem of object aliasing alluded to above in the `document.write` example where multiple variable names refer to the same heap object, to be able to soundly enforce the policies mentioned above, GATEKEEPER needs to statically reason about the program heap. To this end, this paper proposes the first points-to analysis for JavaScript. The programming language community has long recognized pointer analysis to be a key building blocks for reasoning about object-oriented programs. As a result, pointer analyses have been developed commonly used languages such as C and Java, but nothing has been proposed for JavaScript thus far. However, a *sound* and precise points-to analysis of the *full* JavaScript language is very hard to construct. Therefore, we propose a pointer analysis for JavaScript$_{\text{SAFE}}$, a realistic subset that includes prototypes and reflective language

constructs. To handle programs outside of the JavaScript$_{\text{SAFE}}$ subset, GATEKEEPER inserts runtime checks to preclude dynamic code introduction. Both the pointer analysis and nine policies we formulate on top of the points-to results are written on top of the same expressive Datalog-based declarative analysis framework. As a consequence, the hosting site interested in enforcing a security policy can program their policy in several lines of Datalog and apply it to all newly submitted widgets.

In this paper we demonstrate that, in fact, JavaScript programs are far more amenable to analysis than previously believed. To justify our design choices, we have evaluated over 8,000 JavaScript widgets, from sources such as Live.com, Google, and the Vista Sidebar. Unlike some previous proposals [28], JavaScript$_{\text{SAFE}}$ is entirely pragmatic, driven by what is found in real-life JavaScript widgets. Encouragingly, we have discovered that the use of `with`, `Function` and other "difficult" constructs [10] is similarly rare. In fact, `eval`, a reflective construct that usually foils static analysis, is only used in 6% of our benchmarks. However, statically unknown field references such `a[index]`, dangerous because these can be used to get to `eval` through `this['eval']`, etc., and `innerHtml` assignments, dangerous because these can be used to inject JavaScript into the DOM, are more prevalent than previously thought. Since these features are quite common, to prevent runtime code introduction and maintain the soundness of our approach, GATEKEEPER inserts dynamic checks around statically unresolved field references and `innerHtml` assignments.

This paper contains a comprehensive large-scale experimental evaluation. To show the practicality of GATEKEEPER, we present nine representative policies for security and reliability. Our policies include restricting widgets capabilities to prevent calls to `alert` and the use of the `XmlHttpRequest` object, looking for global namespace pollution, detecting browser redirects leading to cross-site scripting, preventing code injection, taint checking, etc. We experimented on 8,379 widgets, out of which 6,541 are analyzable by GATEKEEPER [1]. Checking our nine policies resulted in us discovering a total of 1,341 verified warnings that affect 684 widgets, with only 113 false positives affecting only two widgets.

1.1 Contributions

This paper makes the following contributions:

- We propose the first points-to analysis for JavaScript programs. Our analysis is the first to handle a prototype-based language such as JavaScript. We also identify JavaScript$_{\text{SAFE}}$, a statically analyzable subset of the JavaScript

[1] Because we cannot ensure soundness for the remaining 1,845 widgets, we reject them without further policy checking.

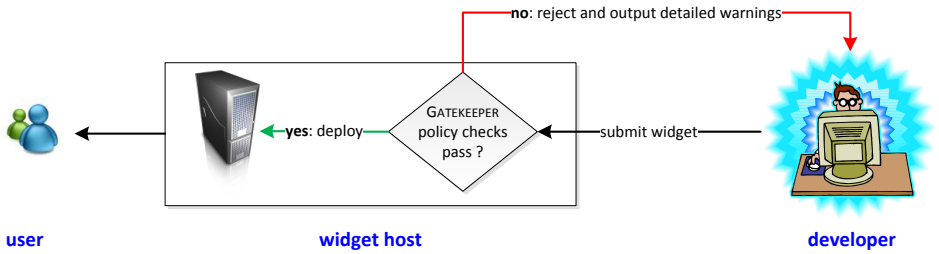

Figure 1: GATEKEEPER deployment. The three principals are: the *user*, the *widget host*, and the *widget developer*.

language and propose lightweight instrumentation that restricts runtime code introduction to handle many more programs outside of the $\text{JavaScript}_{\text{SAFE}}$ subset.

- On the basis of points-to information, we demonstrate the utility of our approach by describing nine representative security and reliability policies that are soundly checked by GATEKEEPER, meaning no false negatives are introduced. These policies are expressed in the form of succinct declarative Datalog queries. The system is highly extensible and easy to use: each policy we present is only several lines of Datalog. Policies we describe include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, etc.
- Our experimental evaluation involves in excess of *eight thousand* publicly available JavaScript widgets from Live.com, the Vista Sidebar, and Google. We flag a total of 1,341 policy violations spanning 684 widgets, with 113 false positives affecting only two widgets.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of our approach and summarizes the most significant analysis challenges. Section 3 provides a deep dive into the details of our analysis; a reader interested in learning about the security policies may skip this section on the first reading. Section 4 describes nine static checkers we have developed for checking security policies of JavaScript widgets. Section 5 summarizes the experimental results. Finally, Sections 6 and 7 describe related work and conclude.

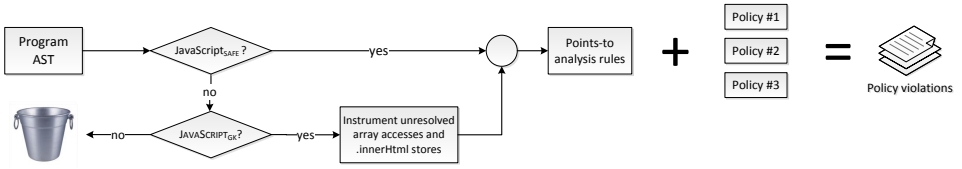

Figure 2: GATEKEEPER analysis architecture.

2 Overview

As a recent report from Finjan Security shows, widget-based attacks are on the rise [13]. Exploits such those in a Vista sidebar contacts widget, a Live.com RSS widget, and a Yahoo! contact widget [13, 21] not only affect unsuspecting users, they also reflect poorly on the hosting site. In a way, widgets are like operating system drivers: their quality directly affects the perceived quality of the underlying OS. While driver reliability and security has been subject of much work [5], widget security has received relatively little attention. Just like with drivers, however, widgets can run in the same page (analogous to an OS process) as the rest of the hosting site. Because widget flaws can negatively impact the rest of the site, it is our aim to develop tools to ensure widget security and reliability.

While our proposed static analysis techniques are much more general and can be used for purposes as diverse as program optimization, concrete type inference, and bug finding, the focus of this paper is on soundly enforcing security and reliability policies of JavaScript widgets. There are three principals that emerge in that scenario: the widget hosting site such as Live.com, the developer submitting a particular widget, and the user on whose computer the widget is ultimately executed. The relationship of these principals is shown in Figure 1. We are primarily interested in helping the *widget host* ensure that their users are protected.

2.1 Deployment

We envision GATEKEEPER being deployed and run by the widget hosting provider as a mandatory checking step in the online submission process, required before a widget is accepted from a widget developer. Many hosts already use captchas to ensure that the submitter is human. However, captchas say nothing about the quality and intent of the code being submitted. Using GATEKEEPER will ensure that the widget being submitted complies with the policies chosen by the host. A hosting provider has the authority to reject some of the submitted widgets, instructing widgets authors to change their code until it passes the policy checker, not unlike tools like the static driver verifier for Windows drivers [18]. Our policy checker

JavaScript Construct	Sidebar Affected	Sidebar %	Windows Live Affected	Windows Live %	Google Affected	Google %
Non-Const Index	1,736	38.6%	176	6.5%	192	16.4%
`with`	422	9.4%	2	.1%	2	.2%
`arguments`	175	3.9%	6	.2%	3	.3%
`setTimeout`	824	18.3%	49	1.8%	65	5.6%
`setInterval`	377	8.4%	16	.6%	13	1.1%
`eval`	353	7.8%	10	.4%	55	4.7%
`apply`	173	3.8%	29	1.1%	6	.5%
`call`	151	3.4%	2,687	99.0%	4	.3%
`Function`	142	3.2%	4	.1%	21	1.8%
`document.write`	102	2.3%	1	0%	108	9.2%
`.innerHTML`	1,535	34.1%	2,053	75.6%	288	24.6%

Figure 3: Statistics for 4,501 widgets from Sidebar and 2,714 widgets from Live, and 1,171 widgets from Google.

outputs detailed information about why a particular widget fails, annotated with line numbers, which allows the widget developer to fix their code and resubmit.

2.2 Designing Static Language Restrictions

To enable sound analysis, we first restrict the input to be a subset of JavaScript as defined by EcmaScript-262 language standard. Unlike previous proposals that significantly hamper the language expressiveness for the sake of safety [9], our restrictions are relatively minor. In particular, we disallow the `eval` construct and its close cousin, the `Function` object constructor as well as functions `setTimeout` and `setInterval`. All of these constructs take a string and execute it as JavaScript code. The fundamental problem with these constructs is that they introduce new code at runtime that is unseen — and cannot be reasoned about — by the static analyzer. These constructs have the same expressive power: allowing one of them is enough to have the possibility of arbitrary code introduction.

We also disallow the use of `with`, a language feature that allows to dynamically substitute the symbol lookup scope, a feature that has few legitimate uses and significantly complicates static reasoning about the code. As our treatment of prototypes shows, it is in fact possible to handle `with`, but it is only used in 8% of our benchmarks.

We do allow reflective constructs `Function.call`, `Function.apply`, and the `arguments` array. Indeed, `Function.call`, the construct that allows the caller of a function to set the callee's `this` parameter, is used in 99% of Live widgets and can be analyzed statically with relative ease, so we handle this language feature. In

Feature	JavaScript$_{\text{SAFE}}$	JavaScript$_{\text{GK}}$
UNCONTROLLED CODE INJECTION		
Unrestricted `eval`	✗	✗
`Function` constructor	✗	✗
`setTimeout`, `setInterval`	✗	✗
`with`	✗	✗
`document.write`	✗	✗
Stores to code-injecting fields `innerHtml`, `onclick`, etc.	✗	✗
CONTROLLED REFLECTION		
`Function.call`	✓	✓
`Function.apply`	✓	✓
`arguments` array	✓	✓
INSTRUMENTATION POINTS		
Non-static field stores	✗	✓
`innerHtml` assignments	✗	✓

Figure 4: Support for different dynamic EcmaScript-262 language features in JavaScript$_{\text{SAFE}}$ and JavaScript$_{\text{GK}}$ language subsets.

other words, our analysis choices are driven by the statistics we collect from 8,379 real-world widgets and not hypothetical considerations. More information about the relative prevalence of "dangerous" language features can be found in Figure 3. The most common "unsafe" features we have to address are `.innerHtml` assignments and statically unresolved field references. Because they are so common, we cannot simply disallow them, so we check them at runtime instead.

To implement restrictions on the allowed input, in our JavaScript parser we flag the use of tokens `eval`, `Function`, and `with`, as well as `setTimeout`, and `setInterval`. We need to disallow all of these constructs because letting one of them through is enough for arbitrary code introduction. The feature we cannot handle simply using token blacklisting is `document.write`. We first optimistically assume that no calls to `document.write` are present and then proceed to verify this assumption as described in Section 4.3. This way our analysis remains sound.

We consider two subsets of the JavaScript language, JavaScript$_{\text{SAFE}}$ and JavaScript$_{\text{GK}}$. The two subsets are compared in Figure 4. If the program passes the checks above *and* lacks statically unresolved array accesses and `innerHtml` assignments, it is declared to be in JavaScript$_{\text{SAFE}}$. Otherwise, these dangerous accesses are instrumented and it is declared in the JavaScript$_{\text{GK}}$ language subset. To resolve field accesses, we run a local dataflow constant propagation analysis [1] to

identify the use of constants as field names. In other words, in the following code snippet

```
var fieldName = 'f';
a[fieldName] = 3;
```

the second line will be correctly converted into `a.f = 3`.

2.3 Analysis Process

The analysis process is summarized in Figure 2. If the program is outside of JavaScript$_{\text{GK}}$, we reject it right away. Otherwise, we first traverse the program representation and output a database of facts, expressed in Datalog notation. This is basically a declarative database representing what we need to know about the input JavaScript program. We next combine these facts with a representation of the native environment of the browser discussed in Section 3.5 and the points-to analysis rules. All three are represented in Datalog and can be easily combined. We pass the result to bddbddb, an off-the-shelf declarative solver [26], to produce policy violations. This provides for a very agile experience, as changing the policy usually only involves editing several lines of Datalog.

2.4 Processing JavaScript$_{\text{SAFE}}$

For a JavaScript$_{\text{SAFE}}$ program, we normalize each function to a set of statements shown in Figure 5. Note that the JavaScript$_{\text{SAFE}}$ language, which we shall extend in Section 3 is very much Java-like and is therefore amenable to inclusion-based points-to analysis [26]. What is not made explicit by the syntax is that JavaScript$_{\text{SAFE}}$ is a prototype-based language, not a class-based one. This means that objects do not belong to explicitly declared classes. Instead, a object creation can be based on a function, which becomes that object's prototype. Furthermore, we support a restricted form of reflection including `Function.call`, `Function.apply`, and the `arguments` array. The details of pointer analysis are captured in the Datalog rules in Figure 7 and discussed in detail in Section 3.

The crux of the GATEKEEPER approach is to perform declarative analysis on top of the program representation computed via the points-to analysis. One key distinction of our approach with Java is that there is basically no distinction of heap-allocation objects and function closures in the way the analysis treats them. In other words, at a call site, if the base of a call "points to" an allocation site that corresponds to a function declaration, we statically conclude that that function might be called. While it may be possible to recover portions of the call graph

$s ::=$	
ϵ	[EMPTY]
$s; s$	[SEQUENCE]
$v_1 = v_2$	[ASSIGNMENT]
$v = \bot$	[PRIMASSIGNMENT]
`return` $v;$	[RETURN]
$v =$ `new` $v_0(v_1, ..., v_n);$	[CONSTRUCTOR]
$v = v_0(v_1, v_2, \ldots, v_n);$	[CALL]
$v_1 = v_2.f;$	[LOAD]
$v_1.f = v_2;$	[STORE]
$v =$ `function`$(v_1, ..., v_n)\ \{s;\};$	[FUNCTIONDECL]

Figure 5: JavaScript$_{\text{SAFE}}$ statement syntax in BNF.

through local analysis, we interleave call graph and points-to analysis in our approach.

We are primarily concerned with analyzing objects or references to them in the JavaScript heap and not primitive values such as integers and strings. We therefore do not attempt to faithfully model primitive value manipulation, lumping primitive values into PRIMASSIGNMENT statements.

2.5 Analysis Soundness

The core static analysis implemented by GATEKEEPER is sound, meaning that we statically provide a conservative approximation of the runtime program behavior. Achieving this for JavaScript with all its dynamic features is far from easy. As a consequence, we extend our soundness guarantees to programs utilizing a smaller subset of the language. For programs within JavaScript$_{\text{SAFE}}$, our analysis is sound. For programs within GATEKEEPER, our analysis is sound *as long as no code introduction is detected with the runtime instrumentation we inject*. This is very similar to saying that, for instance, a Java program is not going to access outside the boundaries of an array as long as no `ArrayOutOfBoundsException` is thrown. Details of runtime instrumentation are presented in Section 3.3. The implications of soundness is that GATEKEEPER is guaranteed to flag all policy violations, at the cost of potential false positives.

We should also point out that the GATEKEEPER analysis is inherently a *whole-program analysis*, not a modular one. The need to statically have access to the entire program is why we work so hard to limit language features that allow dy-

$v_1 = v_2$	$\textsc{Assign}(v_1, v_2)$.	[ASSIGNMENT]
$v = \bot$		[BOTASSIGNMENT]
`return` v	$\textsc{CallRet}(v)$.	[RETURN]
$v = \texttt{new}\ v_0(v_1, v_2, ..., v_n)$	$\textsc{PtsTo}(v, d_{fresh})$. $\textsc{Prototype}(d_{fresh}, h) :-$ $\quad \textsc{PtsTo}(v_0, m)$, $\quad \textsc{HeapPtsTo}(m, \texttt{"prototype"}, h)$. $\forall z > 0 : \textsc{Actual}(i, z, v_z)$. $\forall z > 0 : \textsc{CallRet}(i, v)$.	[CONSTRUCTOR]
$v = v_0(v_1, v_2, \ldots, v_n)$	$\forall z > 0 : \textsc{Actual}(i, z, v_z)$. $\forall z > 0 : \textsc{CallRet}(i, v)$.	[CALL]
$v_1 = v_2.f$	$\textsc{Load}(v_1, v_2, f)$.	[LOAD]
$v_1.f = v_2$	$\textsc{Store}(v_1, f, v_2)$.	[STORE]
$v = \texttt{function}(v_1, ..., v_n)\ \{s\}$	$\textsc{PtsTo}(v, d_{fresh})$. $\textsc{HeapPtsTo}(d_{fresh},$ $\quad \texttt{"prototype"}, p_{fresh})$. $\textsc{FuncDecl}(d_{fresh})$. $\textsc{Prototype}(p_{fresh}, h_{FP})$. $\textsc{MethodRet}(d_{fresh}, v)$. $\textsc{Formal}(d_{fresh}, z, v_z)$.	[FUNCTIONDECL]

Figure 6: Datalog facts generated for each JavaScript$_{\text{SAFE}}$ statement.

namic code loading or injection.

3 Analysis Details

This section is organized as follows. Section 3.1 talks about pointer analysis in detail. Section 3.3 discusses the runtime instrumentation inserted by GATEKEEPER. Section 3.4 talks about how we normalize JavaScript AST to fit into our intermediate representation. Section 3.5 talks about how we model the native JavaScript environment.

% Basic rules		
$\textsc{PtsTo}(v, h)$	:–	$\textsc{Alloc}(v, h)$.
$\textsc{PtsTo}(v, h)$	:–	$\textsc{FuncDecl}(v, h)$.
$\textsc{PtsTo}(v_1, h)$	:–	$\textsc{PtsTo}(v_2, h), \textsc{Assign}(v_1, v_2)$.
$\textsc{DirectHeapStoresTo}(h_1, f, h_2)$	:–	$\textsc{Store}(v_1, f, v_2), \textsc{PtsTo}(v_1, h_1), \textsc{PtsTo}(v_2, h_2)$.
$\textsc{DirectHeapPointsTo}(h_1, f, h_2)$	:–	$\textsc{DirectHeapStoresTo}(h_1, f, h_2)$.
$\textsc{PtsTo}(v_2, h_2)$	:–	$\textsc{Load}(v_2, v_1, f), \textsc{PtsTo}(v_1, h_1),$ $\textsc{HeapPtsTo}(h_1, f, h_2)$.
$\textsc{HeapPtsTo}(h_1, f, h_2)$	:–	$\textsc{DirectHeapPointsTo}(h_1, f, h_2)$.
% Call graph		
$\textsc{Calls}(i, m)$	:–	$\textsc{Actual}(i, 0, c), \textsc{PtsTo}(c, m)$.
% Interprocedural assignments		
$\textsc{Assign}(v_1, v_2)$	:–	$\textsc{Calls}(i, m), \textsc{Formal}(m, z, v_1),$ $\textsc{Actual}(i, z, v_2), z > 0$.
$\textsc{Assign}(v_2, v_1)$	:–	$\textsc{Calls}(i, m), \textsc{MethodRet}(m, v_1),$ $\textsc{CallRet}(i, v_2)$.
% Prototype handling		
$\textsc{HeapPtsTo}(h_1, f, h_2)$	:–	$\textsc{Prototype}(h_1, h), \textsc{HeapPtsTo}(h, f, h_2)$.

Figure 7: Pointer analysis inference rules for JavaScript$_{\text{SAFE}}$ expressed in Datalog.

3.1 Pointer Analysis

We define the following *domains* for the points-to analysis GATEKEEPER performs: heap-allocated objects and functions H, program variables V, call sites I, fields F, and integers Z. Note that the potentially unbounded number of heap-allocated objects is approximated using a statically fixed number of allocation sites in the program. The analysis operates on a number of relations of fixed arity and type. In particular,

- $\textsc{Calls}(i : I, m : H)$ holds when call site i invokes method m.
- $\textsc{Formal}(m : H, z : Z, v : V)$ and $\textsc{MethodRet}(m : H, v : V)$ record formals of a function as well as its return value. $\textsc{Formal}(m, z, v)$ means that function m has z-th formal parameter v. $\textsc{MethodRet}(m, v)$ means that method m's return parameter is v.
- $\textsc{Actual}(i : I, z : Z, v : V)$ and $\textsc{CallRet}(i : I, v : V)$ record actuals of a function call as well as the return value. $\textsc{Actual}(i, z, v)$ means that at call site i, z-th actual is v.
- $\textsc{DeclaredIn}(i : I, m : H)$ means that call site i is located in function m.

- ASSIGN$(v_1 : V, v_2 : V)$ records variable assignment of the form $v_1 = v_2$.
- LOAD$(v_1 : V, v_2 : V, f : F)$ and STORE$(v_1 : V, f : F, v_2 : V)$ represent field loads and stores for heap-based objects. LOAD(v_1, v_2, f) corresponds to the load operation $v_1 = v_2.f$. STORE$(v_1 : V, f : F, v_2 : V)$ corresponds to the store operation $v_1.f = v_2$.
- PTSTO$(v : V, h : H)$ and HEAPPTSTO$(h_1 : H, f : F, h_2 : H)$ represent points-to relations for variables and heap objects, respectively. PTSTO(v, h) means that variable v may point to heap-allocated object h. HEAPPTSTO(h_1, f, h_2) means that field f of object h_1 may point to object h_2.
- Finally, the PROTOTYPE$(h_1 : H, h_2 : H)$ relation records a static approximation of object prototypes. PROTOTYPE(h_1, h_2) means that the implicit prototype for object h_1 may be h_2.

Starting with a set of initial input relation, the analysis follows inference rules, updating intermediate relation values until a fixed point is reached. Details of declarative analysis and BDD-based representation can be found in [25]. The analysis proceeds in stages. In the first analysis stage, we traverse the normalized representation for JavaScript$_{\text{SAFE}}$ shown in Figure 5. The basic facts that are produced for every statement in the JavaScript$_{\text{SAFE}}$ program are summarized in Figure 6. As part of this traversal, we fill in relations ASSIGN, FORMAL, ACTUAL, METHODRET, CALLRET, etc. This is a relatively standard way to represent information about the program in the form of a database of facts. The second stage applies Datalog inference rules to the initial set of facts. The analysis rules are summarized in Figure 7. In the rest of this section, we discuss different aspects of the pointer analysis.

3.1.1 Call Graph Construction

Call graph construction in JavaScript presents a number of challenges. First, unlike a language with function pointers like C, or a language with a fixed class hierarchy like Java, JavaScript does *not* have any initial call graph to start with. Aside from local analysis, the only conservative default we have to fall back to when doing static analysis is "any call site calls every declared function," which is too imprecise.

Instead, we chose to combine points-to and call graph constraints into a single Datalog constraint system and resolve them at once. Informally, intraprocedural data flow constraints lead to new edges in the call graph. These in turn lead to new data flow edges when we introduce constraints between newly discovered arguments and return values. In a sense, function declarations and object allocation sites are treated very much the same in our analysis. If a variable $v \in V$ may point

```
1. function T(){ this.foo = function(){ return 0;}};   d_T, p_T
2. var t = new T();                                    a_1
3. T.prototype.bar = function(){ return 1; };          d_bar, p_bar
4. t.bar(); // return 1
```

Figure 8: Prototype manipulation example.

to function declaration f, this implies that call $v()$ may invoke function f. Allocation sites and function declarations flow into the points-to relation PTSTO through relations ALLOC and FUNCDECL.

3.1.2 Prototype Treatment

The JavaScript language defines two lookup chains. The first is the lexical (or static) lookup chain common to all closure-based languages. The second is the prototype chain. To resolve `o.f`, we follow `o`'s prototype, `o`'s prototype's prototype, etc. to locate the first object associated with field `f`.

Note that the object prototype (sometimes denoted as [[`Prototype`]]) is different the `prototype` field available on any object. We model [[`Prototype`]] through the PROTOTYPE relation in our static analysis. When PROTOTYPE(h_1, h_2) holds, h_1's internal [[`Prototype`]] may be h_2 [2].

Two rules in Figure 6 are particularly relevant for prototype handling: [CONSTRUCTOR] and [FUNCTIONDECL]. In the case of a constructor call, we allocate a new heap variable d_{fresh} and make the return result of the call v point to it. For (every) function m the constructor call invokes, we make sure that m's `prototype` field is connected with d_{fresh} through the PROTOTYPE relation. We also set up ACTUAL and CALLRET values appropriately. In the case of a [FUNCTIONDECL], we create two fresh allocation site, d_{fresh} for the function and p_{fresh} for the newly create `prototype` field for that function. We use shorthand notion h_{FP} to denote object `Function.prototype` and create a PROTOTYPE relation between p_{fresh} and h_{FP}. We also set up HEAPPTSTO relation between d_{fresh} and p_{fresh} objects. Finally, we set up relations FORMAL and METHODRET.

Example 1. The example in Figure 8 illustrates the intricacies of prototype manipulation. Allocation site a_1 is created on line 2. Every declaration creates a declaration object and a prototype object, such as d_T and p_T. The following rules are output in as this code is processed, annotated with the line number they come from:

[2] We follow the EcmaScript-262 standard; Firefox makes [[`Prototype`]] accessible through a non-standard field `__proto__`.

1. PTSTO(T, d_T). HEAPPTSTO(d_T, "prototype", p_T). PROTOTYPE($p_T, h_{\texttt{FP}}$).
2. PTSTO(t, a_1). PROTOTYPE(a_1, p_T).
3. HEAPPTSTO(p_T, "bar", $d_{\texttt{bar}}$). HEAPPTSTO($d_{\texttt{bar}}$, "prototype", $p_{\texttt{bar}}$). PROTOTYPE($p_{\texttt{bar}}, h_{\texttt{FP}}$).

To resolve the call on line 4, we need to determine what `t.bar` points to. Given PTSTO(t, a_1) on line 2, this resolves to the following Datalog query:

$$\text{HEAPPTSTO}(a_1, \texttt{"bar"}, X)?$$

Since there is nothing d_T points to *directly* by following the `bar` field, the PROTOTYPE relation is consulted. PROTOTYPE(a_1, p_T) comes from line 2. Because we have HEAPPTSTO(p_T, "bar", $d_{\texttt{bar}}$) on line 3, we resolve X to be $d_{\texttt{bar}}$. As a result, the call on line 4 may correctly invoke function `bar`. Note that our rules do not try to keep track of the order of objects in the prototype chain. □

3.2 Handling Reflection

We add the following rules to handle reflective call constructs to the rules in Figure 7:

$$v = v_0.\texttt{apply}(v_{this}, [v_1, v_2, \ldots, v_n]) \quad \text{[APPLYREFLECTIVE]}$$
$$v = v_0.\texttt{call}(v_{this}, v_1, v_2, \ldots, v_n) \quad \text{[CALLREFLECTIVE]}$$

The rules to handle these reflective invokations as as follows for `apply`, where invocation site i' is a fresh call site at the same code location as the orginal reflective call to `apply`:

$$\begin{aligned}
&\text{ACTUAL}(i, z, v) \;:-\; GlobalSym(\texttt{"Function"}, fun), \\
&\quad \text{HEAPPTSTO}(fun, \texttt{"apply"}, a), \text{CALLS}(i, a), \\
&\quad \text{ACTUAL}(i, z+1, v), z < 2. \\
&\text{ACTUAL}(i, 1, \texttt{"global"}) \;:-\; GlobalSym(\texttt{"Function"}, fun), \\
&\quad \text{HEAPPTSTO}(fun, \texttt{"apply"}, a), \text{CALLS}(i, a). \\
&\text{ACTUAL}(i, z, v) \;:-\; GlobalSym(\texttt{"Function"}, fun), \\
&\quad \text{HEAPPTSTO}(fun, \texttt{"apply"}, a), \text{CALLS}(i, a), \\
&\quad \text{ACTUAL}(i, 3, v'), \text{PTSTO}(v', h), \\
&\quad \text{HEAPPTSTO}(h, f, h'), \\
&\quad \text{PTSTO}(v, h'), Num2Str(z-2, f), z > 1. \\
&\text{CALLRET}(i, v) \;:-\; GlobalSym(\texttt{"Function"}, fun), \\
&\quad \text{HEAPPTSTO}(fun, \texttt{"apply"}, a), \text{CALLS}(i, a), \\
&\quad \text{CALLRET}(i, v).
\end{aligned}$$

Function `call` is addressed similarly, except ACTUAL values are shifted by one:

$$\begin{array}{lll}
\textsc{Actual}(i, z, v) & :- & \mathit{GlobalSym}(\texttt{"Function"}, \mathit{fun}), \\
& & \textsc{HeapPtsTo}(\mathit{fun}, \texttt{"call"}, a), \textsc{Calls}(i, a), \\
& & \textsc{Actual}(i, z+1, v). \\
\textsc{Actual}(i, 1, \texttt{"global"}) & :- & \mathit{GlobalSym}(\texttt{"Function"}, \mathit{fun}), \\
& & \textsc{HeapPtsTo}(\mathit{fun}, \texttt{"call"}, a), \textsc{Calls}(i, a). \\
\textsc{CallRet}(i, v) & :- & \mathit{GlobalSym}(\texttt{"Function"}, \mathit{fun}), \\
& & \textsc{HeapPtsTo}(\mathit{fun}, \texttt{"call"}, a), \textsc{Calls}(i, a), \\
& & \textsc{CallRet}(i, v).
\end{array}$$

The next challenge is to handle the `arguments` array. In addition to actual parameters for the current function, it also has elements `caller` and `callee` referring to the calling and the current function, respectively. The actual parameters are handled the following way. When processing a [FUNCTIONDECL], we add the following relation

$$\begin{array}{l}
\textsc{PtsTo}(\texttt{"arguments"}, a_{\mathit{fresh}}). \\
\textsc{Arguments}(d_{\mathit{fresh}}, a_{\mathit{fresh}}).
\end{array}$$

This effectively declares a new `arguments` object for each function and connects the allocation site for the function declaration to that object using relation $\textsc{Arguments} : H \times H$. Formals of the function and `arguments` elements are connected the following way:

$$\begin{array}{lll}
\textsc{DirectHeapStoresTo}(a, f_i, h) & :- & \textsc{Arguments}(d, a), \textsc{Formal}(d, i+2, v), \\
& & \textsc{PtsTo}(v, h), \textsc{Int2Str}(i, f_i).
\end{array}$$

where INT2STR is an auxiliary relation used to convert between integers and strings. It is set up by declaring $\textsc{Int2Str}(0, \texttt{"0"})$. $\textsc{Int2Str}(1, \texttt{"1"})$. $\textsc{Int2Str}(2, \texttt{"2"}) \ldots$. The numbering offset of 2 is needed because `arguments`$[0]$ is the same as formal number 2 (formal number 1 is the `this` parameter for the current function). Next, the `callee` field of the arguments array is set up the following way:

$$\textsc{DirectHeapStoresTo}(a, \texttt{"callee"}, d) :- \textsc{Arguments}(d, a).$$

The `caller` field is set to any potential caller of function d as follows:

$$\begin{array}{lll}
\textsc{DirectHeapStoresTo}(a, \texttt{"caller"}, d') & :- & \textsc{Calls}(i, d), \textsc{Arguments}(d, a), \\
& & \textsc{DeclaredIn}(i, d').
\end{array}$$

Recall that relation DECLAREDIN records the function in which call site is located.

3.3 Rewriting GATEKEEPER Programs Outside JavaScript$_{\text{SAFE}}$

The focus of this section is on runtime instrumentation for programs outside JavaScript$_{\text{SAFE}}$, but within the JavaScript$_{\text{GK}}$ JavaScript subset that is designed to prevent runtime code introduction.

3.3.1 Rewriting Unresolved Heap Loads and Stores

Syntax for JavaScript$_{\text{GK}}$ supported by GATEKEEPER has an extra variant of LOAD and STORE rules for associative arrays, which introduce Datalog facts shown below:

$$v_1 = v_2[*] \quad \textsc{Load}(v_1, v_2, \_) \quad [\textsc{ArrayLoad}]$$
$$v_1[*] = v_2 \quad \textsc{Store}(v_1, \_, v_2) \quad [\textsc{ArrayStore}]$$

When the indices of an associative array operation cannot be determined statically, we have to be conservative. This means that any field that may potentially be reached should be considered as accessed.

Example 2. Consider the following motivating example:

```
1. var a = {'f' : function(){...}, 'g' : function(){...}, ...};
2. a[x + y] = function(){...};
3. a.f();
```

If we cannot statically decide which field of object `a` is being written to on line 2, we have to conservatively assume that the assignment could be to field `f`. This can affect which function is called on line 3. □

Moreover, any statically unresolved store may introduce additional code through writing to the `innerHtml` field that will be never seen by static analysis. We rewrite statically unsafe stores $v_1[i] = v_2$ by blacklisting field that may lead to code introduction:

```
if ( (i=="innerHtml" || i.toLowerCase()=="onclick" || ...)
    && __IsUnsafe(v2))
{
    abort("Disguised eval attempt at <file>:<line>");
} else {
    v1[i] = v2;
}
```

where function `__IsUnsafe` disallows all but very simple HTML. Similarly, statically unsafe loads of the form $v_1 = v_2[i]$ can be restricted as follows:

```
if (i=="eval" || i=="setInterval" || i=="setTimeout" ||...) {
    abort("Disguised eval attempt at <file>:<line>");
} else {
    v1 = v2[i];
}
```

Note that we have to check for unsafe functions such as `eval`, `setInterval`, etc. While we reject them as tokens for JavaScript$_{\text{SAFE}}$, they can still try to creep in through the use of statically unresolved array accesses. As an alternative to explicit runtime checking, we could wrap i in a call to `toSafeHtml`, a construct supported in newer versions of Internet Explorer, but not yet universally adopted.

Note that to preserve the soundness of our analysis, care must be taken to blacklist all the ways to inject code. We do our best to ensure that all the relevant fields are checked for and our results are sound assuming we eliminate all the possibilities.

3.3.2 Rewriting `.innerHtml` assignments

`innerHtml` assignments are a common dangerous language feature that may prevent GATEKEEPER from statically seeing all the code. We disallow it in JavaScript$_{\text{SAFE}}$, but because it is so common, we still allow it in the JavaScript$_{\text{GK}}$ language subset. While in many cases the right-hand side of `.innerHtml` assignments is a constant, there is an unfortunate coding pattern encouraged by Live widgets that makes static analysis difficult:

```
this.writeWidget = function(widgetURL) {
    var url = "http://widgets.clearspring.com/csproduct/web/show/flash?
        opt=-MAX/1/-PUR/http%253A%252F%252Fwww.microsoft.com&url="+widgetURL;

    var myFrame = document.createElement("div");
    myFrame.innerHTML = '<iframe id="widgetIFrame" scrolling="no"
        frameborder="0" style="width:100%;height:100%;border:0px" src="'+
        url+'"></iframe>';
    ...
}
```

The `url` value, which is the result concatenating of a constant URL and `widgetURL` is being used on the right-hand side and could be used for code injection. An assignment v_1`.innerHtml` $= v_2$ is rewritten as

```
if (__IsUnsafe(v2)) {
    abort("Disguised eval attempt at <file>:<line>");
} else {
    v1.innerHtml = v2;
}
```

3.4 Normalization Details

In this section we discuss various aspects of normalizing the JavaScript AST.

Handling the global object. We treat the global object explicitly by introducing a variable `global` and then assigning to its fields. One interesting detail is that global variable reads and writes become *loads* and *stores* to fields of the global object, respectively.

Handling of `this` argument in function calls. One curious feature of JavaScript is its treatment of the `this` keyword, which is described in section 10.2 of the EcmaScript-262 standard. For calls of the form $\mathtt{f}(\mathtt{x}, \mathtt{y}, ...)$, the `this` value is set by the runtime to the global object. This is a pretty surprising design choice, so we translate syntactic forms $\mathtt{f}(\mathtt{x}, \mathtt{y}, ...)$ and $\mathtt{o.f}(\mathtt{x}, \mathtt{y}, ...)$ differently, passing the global object in place of `this` in the former case. When translating the program into Datalog, argument `this` is made parts of the ACTUAL rule: we explicitly pass it as actual argument number 1.

Object literals. The object literal $\mathtt{v} = \{\mathtt{f}_1 : \mathtt{v}_1, \mathtt{f}_2 : \mathtt{v}_2, \ldots\}$ notation in JavaScript can be interpreted as shorthand for $\mathtt{v.f}_1 = \mathtt{v}_1;\ \mathtt{v.f}_2 = \mathtt{v}_2; \ldots$.

3.5 Native Environment

The browser embedding of the JavaScript engine has a large number of predefined objects. In addition to `Array`, `Date`, `String`, and other objects defined by the EcmaScript-262 standard, the browser defines objects such as `Window` and `Document`. Because we are doing whole-program analysis, we need to create stubs for the native environment so that calls to built-in methods resolve to actual functions. We recursively traverse the native embedding. For every function we encounter, we provide a default stub $\mathtt{function}()\{\mathtt{return\ undefined};\}$. The resulting set of declaration looks as follows:

```
var global = new Object();
// this references in the global namespace refer to global
var this = global;
global.Array = new Object();
global.Array.constructor = new function(){return undefined;}
global.Array.join = new function(){return undefined;}
 ...
```

Note that we use an explicit `global` object to host a namespace for our declarations instead of the implicit `this` object that JavaScript uses. In most browser implementations, the global `this` object is aliased with the `window` object, leading to the following declaration: `global.window = global;`.

However, as it turns out, creation of a *sound* native environment is more difficult than that. Indeed, the approach above assumes that the built-in functions return objects that are never aliased. This fallacy is most obviously demonstrated by the following code:

```
var parent_div = document.getElementById('header');
var child_div = document.createElement('div');
parent_div.appendChild(child_div);
var child_div2 = parent_div.childNodes[0];
```

In this case, `child_div` and `child_div2` are aliases for the same DIV element. if we pretend they are not, we will miss an existing alias. We therefore model operations such as `appendChild`, etc. in JavaScript code, effectively creating *mock-ups* instead of native browser-provided implementations.

4 Security and Reliability Policies

This section is organized as follows. Sections 4.1–4.4 talk about six policies that apply to widgets from all widgets hosts we use in this paper (Live, Sidebar, and Google). Section 4.5 talks about host-specific policies, where we present two policies specific to Live and one specific to Sidebar widgets. Along with each policy, we present the Datalog query that is designed to find its violations. We have run these queries on our set of 8,379 benchmark widgets. We summarize our experimental findings in Section 5.

4.1 Restricting Widget Capabilities

Perhaps the most common requirement for a system that reasons about widget is the ability to restrict code capabilities, such as disallowing calling a particular function, using a particular object or namespace, etc. The Live Widget Developer Checklist provides many such examples [27]. This is also what system like Caja and WebSandbox aim to accomplish [19, 22]. We can achieve the same goal statically.

Pop-up boxes represent a major annoyance when using web sites. Widgets that bring up popup boxes, achieved by calling function `alert` in JavaScript, can be used for denial-of-service against the user. In fact, the `alert` box prevention example below comes from a sample of a widget that asynchronously spawns of

new alert boxes, distributed with WebSandbox [20]. The following query ensures that the `alert` routine is never called:

Query output: $AlertCalls(i : I)$

$$\begin{array}{lll} GlobalSym(m, h) & :- & \textsc{PtsTo}(\texttt{"global"}, g), \textsc{HeapPtsTo}(g, m, h). \\ AlertCalls(i) & :- & GlobalSym(\texttt{"alert"}, h), \textsc{Calls}(i, h). \end{array}$$

To define $AlertCalls$, we first define an auxiliary query $GlobalSym : F \times H$ used for looking up global functions such as `alert`. On the right-hand side, $g \in H$ is the explicitly represented global object pointed to by variable `global`. Following field m takes us to the heap object h of interest. $AlertCalls$ instantiates this query for field `alert`. Note that there are several references to it in the default browser environment such as `window.alert` and `document.alert`. Since they all are aliases for the same function, the query above will spot all calls, independently of the the reference being used.

4.2 Detecting Writes to Frozen Objects

We disallow changing properties of built-in functions such as `Array`, `Date`, `Document`, etc. to prevent environment pollution attacks such as prototype hijacking [7]. This is similar to *frozen objects* proposed in EcmaScript 4. The query below looks for attempts to add or update properties of JavaScript built-in objects specified by the auxiliary query $BuiltInObject$, including attempts to change their prototypes:

Query output: $FrozenViolation(v : V)$

$$\begin{array}{l} BuiltInObject(h) :- GlobalSym(\texttt{"Boolean"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Array"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Date"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Function"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Math"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Document"}, h). \\ BuiltInObject(h) :- GlobalSym(\texttt{"Window"}, h). \\ \\ FrozenViolation(v) :- \textsc{Store}(v, \_, \_), \textsc{PtsTo}(v, h), BuiltInObject(h). \end{array}$$

The rules above handle the case of assigning to properties of these built-in objects directly. Often, however, a widget might attempt to assign properties of the `prototype` of an object as in `Function.prototype.apply` $=$ `function`$()\{...\}$.

We can prevent this by first defining a recursive heap reachability relation $Reaches$:

$$\begin{array}{lll} Reaches(h_1, h_2) & :- & \textsc{HeapPtsTo}(h_1, \_, h_2). \\ Reaches(h_1, h_2) & :- & \textsc{HeapPtsTo}(h_1, \_, h'), Reaches(h', h_2). \end{array}$$

and then adding to the $FrozenViolation$ definition:

$$\begin{array}{lll} FrozenViolation(v) & :- & \textsc{Store}(v, \_, \_), \textsc{PtsTo}(v, h'), \\ & & BuiltInObject(h), Reaches(h, h'). \end{array}$$

An example of a typical policy violation from our experiments is shown below:

```
Array.prototype.feed = function(o, s){
    if(!s){s=o;o={};}
    var k,p=s.split(":");
    while(typeof(k=p.shift())!="undefined")o[k]=this.shift();
    return o;
}
```

4.3 Detecting Code Injection

As discussed above, `document.write` is a routine that allows the developer to output arbitrary HTML, thus allowing code injection through the use of `<script>` tags. While verbatim calls to `document.write` can be found using `grep`, it is easy to disguise them through the use of aliasing:

```
var x = document;
var y = x.write;
y("<script>alert('hi');</script>");
```

The query below showcases the power of points-to analysis. In addition to finding the direct calls, the query below will correctly determine that the call to `y` invokes `document.write`.

Query output: $DocumentWrite(i : I)$

$$\begin{array}{lll} DocumentWrite(i) & :- & GlobalSym(\texttt{"document"}, d), \\ & & \textsc{HeapPtsTo}(d, \texttt{"write"}, m), \textsc{Calls}(i, h). \\ DocumentWrite(i) & :- & GlobalSym(\texttt{"document"}, d), \\ & & \textsc{HeapPtsTo}(d, \texttt{"writeln"}, m), \textsc{Calls}(i, h). \end{array}$$

4.4 Redirecting the Browser to a Different Location

JavaScript in the browser has write access to the current page's location, which may be used to redirect the user to a malicious site. Google widget `Google_Calculator` performing such redirection is shown below:

```
window.location = "http://e-r.se/google-calculator/index.htm"
```

Allowing such redirect not only opens the door to phishing widgets luring users to malicious sites, redirects within an iframe also opens the possibility of running code that has not been adequately checked by the hosting site, potentially circumventing policy checking entirely. Another very real possibility is cross-site scripting attacks that involve stealing cookies. Of course, `grep` is not an adequate tool for spotting redirects, both because of the aliasing issue described above and because read access to `window.location` is in fact allowed. Moreover, redirects can take many forms, which we discuss in turn below.

Query output: $LocationAssign(v : V)$

$$
\begin{array}{lll}
LocationAssign(v) & :- & GlobalSym(\texttt{"window"}, h), \textsc{PtsTo}(v, h), \\
 & & \textsc{Store}(\_, \texttt{"location"}, v). \\
LocationAssign(v) & :- & GlobalSym(\texttt{"document"}, h), \textsc{PtsTo}(v, h), \\
 & & \textsc{Store}(\_, \texttt{"location"}, v). \\
LocationAssign(v) & :- & \textsc{PtsTo}(\texttt{"global"}, h), \textsc{PtsTo}(v, h), \\
 & & \textsc{Store}(\_, \texttt{"location"}, v).
\end{array}
$$

Storing to location object's properties:

$$
\begin{array}{lll}
LocationAssign(v) & :- & GlobalSym(h, \texttt{"location"}), \textsc{PtsTo}(v, h), \\
 & & \textsc{Store}(v, \_, \_).
\end{array}
$$

Calling methods on the location object:

Query output: $LocationChange(i : I)$

$$
\begin{array}{lll}
LocationChange(i) & :- & LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"assign"}, h'), \\
 & & \textsc{Calls}(i, h'). \\
LocationChange(i) & :- & LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"reload"}, h'), \\
 & & \textsc{Calls}(i, h'). \\
LocationChange(i) & :- & LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"replace"}, h'), \\
 & & \textsc{Calls}(i, h').
\end{array}
$$

Function `window.open` is another form of redirects:

Query output: $WindowOpen(i : I)$

$$WindowOpen(i) \quad :- \quad WindowObject(h), \textsc{HeapPtsTo}(h, \texttt{"open"}, h'), \textsc{Calls}(i, h').$$

4.5 Host-specific Policies

The policies we have discussed thus far have been relatively generic. In this section, we give examples of policies that are specific to the host site they reside on.

4.5.1 No `XMLHttpRequest` Object Use in Live Widgets

The first policy of this sort comes directly from the Live Web Widget Developer Checklist [27]. Among other rules, they disallow the use of `XMLHttpRequest` object in favor of function `Web.Network.createRequest`. The latter makes sure that the network requests are properly proxied so they can work cross-domain:

Query output: $XMLHttpRequest(i : I)$

$$XMLHttpRequest(i) \quad :- \quad GlobalSym(\texttt{"XMLHttpRequest"}, h), \textsc{Calls}(i, h).$$

4.5.2 Detecting Global Namespace Pollution in Live Widgets

Because web widgets can be deployed on a page with other widgets running within the same JavaScript interpreter, polluting the global namespace, leading to name clashes and unpredictable behavior. This is why hosting providers such as Facebook, Yahoo!, Live, etc. strongly discourage pollution of the global namespace, favoring a module or a namespace approach instead [8] that avoids name collision. We can easily prevent stores to the global scope:

Query output: $GlobalStore(h : H)$

$$GlobalStore(h) \quad :- \quad \textsc{PtsTo}(\texttt{"global"}, g), \textsc{HeapPtsTo}(g, \_, h).$$

Here is an example of a violation of this policy from a Live.com widget:

```
var SearchTag = new String ("Home");
var SearchTagStr = new String ("meta%3ASearch.tag%28%22beginTag+" + SearchTag +"endTag%22%29");
var QnaURL= new String (SearchHostPath /*+ SearchQstateStr */+ SearchTagStr +"&format=rss") ;


// define the constructor for your Gadget (this must match the name in the manifest xml)
Microsoft.LiveQnA.RssGadget = function(p_elSource, p_args, p_namespace) { ... }
```

Because the same widget can be deployed twice within the same interpreter scope with different values of `SearchTag`, this can lead to a date race on the globally declared variable `SearchTagStr`.

Note that our analysis approach is radically different from proposals that advocate language restrictions [9, 22] to protect access to the global object. The difficulty those techniques have to overcome is that the `this` identifier in the global scope will point to the global object. However, disallowing `this` completely makes object-oriented programming difficult. With the whole-program analysis GATEKEEPER implements, we do not have this problem. We are able to distinguish references to `this` that point to the global object (aliased with the `global` variable) from a local reference to `this` within a function.

4.5.3 Tainting Data in Sidebar Widgets

This policy ensures that data from ActiveX controls that may be instantiated by a Sidebar widget does not get passed into System.Shell.execute for direct execution on the user's machine. This is because it is common for ActiveX controls to retrieve unsanitized network data, which is how a published RSS Sidebar exploit operates [21]. There, data obtained from an ActiveX-based RSS control was assigned directly to the `innerHtml` field withing a widget, allowing a cross-site scripting exploit. What we are looking for is demonstrated by the pattern:

```
var o = new ActiveXObject();
var x = o.m();
System.Shell.Execute(x);
```

The Datalog query below looks for instances where the tainted result of a call to method `m` on an ActiveX object is directly passed as an argument to the "sink"

function `System.Shell.Execute`:

Query output: $ActiveXExecute(i : I)$

$$
\begin{array}{lll}
ActiveXObjectCalls(i) & :- & GlobalSym(\texttt{"ActiveXObject"}, h'), \textsc{Calls}(i, h'). \\
ShellExecuteCalls(i) & :- & \textsc{PtsTo}(\texttt{"global"}, h_1), \\
 & & \textsc{HeapPtsTo}(h_1, \texttt{"System"}, h_2), \\
 & & \textsc{HeapPtsTo}(h_2, \texttt{"Shell"}, h_3), \\
 & & \textsc{HeapPtsTo}(h_3, \texttt{"execute"}, h_4), \textsc{Calls}(i, h_4). \\
ActiveXExecute(i) & :- & ActiveXObjectCalls(i), \\
 & & \textsc{CallRet}(i, v), \textsc{PtsTo}(v, h), \\
 & & \textsc{HeapPtsTo}(h, \_, m), \textsc{Calls}(i^\star, m), \\
 & & \textsc{CallRet}(i^\star, r), \textsc{PtsTo}(r, h^\star), \\
 & & ShellExecuteCalls(i'), \\
 & & \textsc{Actual}(i', \_, v'), \textsc{PtsTo}(v', h^\star).
\end{array}
$$

Auxiliary queries *ActiveXObjectCalls* and *ShellExecuteCalls* look for source and sink calls and *ShellExecuteCalls* ties all the constraints together, effectively matching the call pattern described above.

5 Experimental Results

For our experiments, we have downloaded a large number of widgets from widget hosting sites' widget galleries. As mentioned before, we have experimented with widgets from Live.com, the Vista Sidebar, and Google. We automated the download process to save widgets locally for analysis. Once downloaded, we parsed through each widget's manifesto to determine where the relevant JavaScript code resides. This process was slightly different across the widget hosts. In particular, Google widgets tended to embed their JavaScript in HTML, which required us to develop a limited-purpose HTML parser. In the Sidebar case, we had to extract the relevant JavaScript code out of an archive. At the end of this process, we ended up with a total of 8,379 JavaScript files to analyze.

Figure 9 provides aggregate statistics for the widgets we used as benchmarks. For each widget source, we specify the total number of widgets we managed to obtain in column 2. Column 3 shows the average lines-of-code count for every widget. In general, Sidebar widgets tend to be longer and more involved than their Web counterparts, as reflected in the average line of code metric. Note that in addition to every widget's code, at the time of policy checking, we also prepend the native environment constructed as described in Section 3.5. The native environment constitutes 270 lines of non-comment JavaScript code (127 for specifying

Widget Source	Avg. LOC	Count	Widget counts JavaScript$_{GK}$		JavaScript$_{SAFE}$	
Live.com	105	2,707	2,643	97%	643	23%
Vista sidebar	261	4,501	2,946	65%	1,767	39%
Google.com/ig	137	1,171	962	82%	768	65%

Figure 9: Aggregate statistics for widgets from Live portal, Windows Sidebar, and Google portal widget repositories.

the the browser embedding and 143 for specifying built-in objects such as `Array` and `Date`).

Result Summary. A summary of our experimental results in presented in Figure 10. For each policy described in Section 4, we show the the total number of violations across 8,379 benchmarks, and the number of violating benchmarks. The latter two may be different because there could be several violations of a particular query per widget. We also show the percentage of benchmarks for which we find policy violations. As can be seen from the table, overall, policy violations are quite uncommon, with only several percent of widgets affected in each case. Overall, a total of 1,341 policy violations are reported. Section 4.5, we only ran those policies on the appropriate subset of widgets, leaving other table cells blank. To validate the precision of our analysis, we have examined all violations reported by our policies. Doing so took about 10 hours of manual effort. Encouragingly, in most cases, GATEKEEPER results were remarkably precise.

False positives. We should point out that a conservative analysis such as GATEKEEPER is inherently imprecise. Two main sources of false positives in our formulation are prototype handling and arrays. Almost all false positives detected in our experiments occur in the Sidebar widget called `JustMusic.FM`, in file `common.js`. When relying on points-to information, it is common for a single imprecision within static analysis to create numerous "cascading" false positive reports. This is the case here as well. Because of our handling of arrays, the analysis conservatively concludes that certain heap-allocated objects can reach many others by following *any* field of array `a`, as shown below:

```
function MM_preloadImages() { //v3.0
  var d=m_Doc; if(d.images){ if(!d.MM_p) d.MM_p=new Array();
    var i,j=d.MM_p.length,a=MM_preloadImages.arguments; for(i=0; i<a.length; i++)
    if (a[i].indexOf("#")!=0){ d.MM_p[j]=new Image; d.MM_p[j++].src=a[i];}}
}
```

		Live Widgets					Vista Sidebar					Google Widgets				
Query	**Section**	**Viol.**	**Affected**		**FP**	**Affected**	**Viol.**	**Affected**		**FP**	**Affected**	**Viol.**	**Affected**		**FP**	**Affected**
$AlertCalls(i : I)$	4.1	54	29	1.1	0	0	161	84	2.9	0	0	57	35	3.6	0	0
$FrozenViolation(v : V)$	4.2	3	3	0.1	0	0	143	52	1.5	94	1	1	1	0.1	0	0
$DocumentWrite(i : I)$	4.3	5	1	0.0	0	0	175	75	1.7	0	0	158	88	8.1	0	0
$LocationAssign(v : V)$	4.4	3	3	0.1	2	1	157	109	3.8	15	1	9	9	0.7	0	0
$LocationChange(i : I)$	4.4	3	3	0.1	0	0	21	20	0.7	1	1	3	3	0.3	0	0
$WindowOpen(i : I)$	4.4	50	22	0.9	0	0	182	87	3.0	1	1	19	14	1.5	0	0
$XMLHttpRequest(i : I)$	4.5	1	1	0.0	0	0	—	—	—	—	—	—	—	—	—	—
$GlobalStore(v : V)$	4.5	136	45	1.7	0	0	—	—	—	—	—	—	—	—	—	—
$ActiveXExecute(i : I)$	4.5	—	—	—	—	—	0	0	0	0	0	—	—	—	—	—

Figure 10: Experimental result summary for nine policies described in Section 4. Because some policies are host-specific, we only run them on a subset of widgets. "—" indicates experiments that are not applicable.

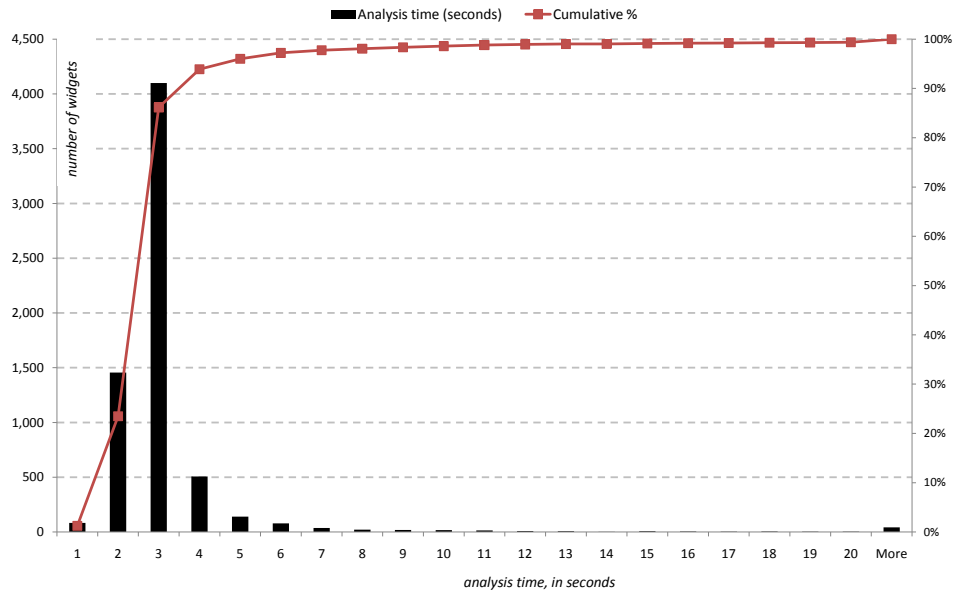

Figure 11: Histogram showing GATEKEEPER processing times, in seconds.

Luckily, it is possible to group cascading reports together in order to avoid overwhelming the user with false positives caused by a single imprecision. This imprecision in turn affects $FrozenViolation$ and $LocationAssign$ queries leading to many very similar reports. A total of 113 false positives are reported, but luckily they affect only two widgets.

Analysis Running Times. Our implementation uses a publicly available declarative analysis engine provided by bddbddb [25]. This is a highly optimized BDD-based solver for Datalog queries used for static analysis in the past. Because repeatedly starting bddbddb is inefficient we perform both the points-to analysis *and* run our Datalog queries corresponding to the policies in Section 4 as part of one run for each widget.

Our analysis is quite scalable in practice, as shown in Figure 11. This histogram shows the distribution of analysis time, in seconds. These results were obtained on a Pentium Core 2 duo 3 GHz machine with 4 GB of memory, running Microsoft Vista SP1. Note that the analysis time includes the JavaScript parsing time, the normalization time, the points-to analysis time, and the time to run all nine policies. For the vast majority of widgets, the analysis time is under 4 seconds, as shown by the cumulative percentage curve in the figure.

Runtime Instrumentation. Programs outside of the $\text{JavaScript}_{\text{SAFE}}$ language sub-

	Live	Sidebar	Google
Number of instrumented files	2,000	1,179	194
Instrumentation points per file	1.74	8.86	5.63

Figure 12: Instrumentation statistics.

set but with the JavaScript$_{GK}$ language subset require instrumentation. Figure 12 summarizes data on the number of instrumentation points required, both as an absolute number and in proportion of the number of widgets that required instrumentation. We plan to assess our runtime overhead as part of future work. However, we do not anticipate it to be very high, as the checks we insert require only several direct pointer comparisons. The number of instrumentation points per instrumented widget ranges roughly in proportion to the size and complexity of the widget. Given the relatively small number of instrumentation points on average and the fact that the runtime check is quite lightweight, we do not anticipate the overhead to be noticeable.

6 Related Work

Much of the work related to this paper focuses on limiting various attack vectors that exist in JavaScript. They do this through the use of type systems, language restrictions, and modifications to the browser or the runtime. We describe these strategies in turn below.

6.1 Static Safety Checks

JavaScript is a highly dynamic language which makes it difficult to reason about programs written in it. However, with certain expressiveness restrictions, desirable security properties can be achieved. ADSafe and Facebook both implement a form of static checking to ensure a form of safety in JavaScript code. ADSafe [9] disallows dynamic content, such as `eval`, and performs static checking to ensure the JavaScript in question is safe. Facebook takes an approach similar to ours in rewriting statically unresolved field stores, however, it appears that, unlike GATEKEEPER, they do not try to do local static analysis of field names. Facebook uses a JavaScript language variant called FBJS [11], that is like JavaScript in many ways, but DOM access is restricted and all variable names are prefixed with a unique identifier to prevent name clashes with other FBJS programs on the same page.

In many ways, however, designing a safe language subset is a tricky business. It is really difficult to write anything but most simple applications in ADSafe because of its static restrictions. On the other hand, while considerably more expressive, FBJS has been subject of several well-publicised attacks that circumvent

the isolation of the global object offered through Facebook sandbox rewriting [2]. This demonstrates that while easy to implement, reasoning about what static language restrictions accomplish is tricky. GATEKEEPER circumvents this problem completely, performing whole program analysis instead. We do no try to prove that $\text{JavaScript}_{\text{SAFE}}$ programs cannot pollute the global namespace, for example. Instead, we take the entire program and a representation of its environment and soundly check if this can happen.

6.2 JavaScript Rewriting and Instrumentation

A practical alternative to static language restrictions is instrumentation. Caja [22] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [19].

Yu et al. traverse the JavaScript document and rewrite based on a security policy [28]. Unlike Caja and WebSandbox, they prove the correctness of their rewriting with operational semantics for a subset of JavaScript called CoreScript. BrowserShield [23] similarly uses dynamic and recursive rewriting to ensure that JavaScript and HTML are safe, for a chosen version of safety, and all content generated by the JavaScript and HTML is also safe. Instrumentation can be used for more than just enforcing security policies. AjaxScope [16] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider.

Compared to these techniques, GATEKEEPER has two clear advantages. First, as a mostly static analysis, GATEKEEPER places little runtime overhead burden on the user. According to some reports, runtime overhead of Caja and WebSandbox can be as high as 5–10x, depending on the level of rewriting. Second, just as with the Facebook exploits mentioned above, it is really difficult to reason about whether source-level rewriting provides complete isolation. We feel that sound static analysis provides a much more systematic way to reason about what code can do.

6.3 Runtime and Browser Support

Current browser infrastructure and the HTML standard require a page to fully trust foreign JavaScript if they want the foreign JavaScript to interact with their site. The alternative is to place foreign JavaScript in an isolated environment, which disallows any interaction with the hosting page. This leads to web sites trusting untrustworthy JavaScript code in order to provide a richer web site. One solu-

tion to get around this all-or-nothing trust problem is to modify browsers and the HTML standard to include a richer security model that allows untrusted JavaScript controlled access to the hosting page.

MashupOS [14] proposes a new browser that is modeled after an OS and modifies the HTML standard to provide new tags that make use of new browser functionality. They provide rich isolation between execution environments, including resource sharing and communication across instances. In a more lightweight modification to the browser and HTML, Felt et al. [12] add a new HTML tag that labels a `div` element as untrusted and limits the actions that any JavaScript inside of it can take. This would allow content providers to create a sand box in which to place untrusted JavaScript. We can even imagine integrating GATEKEEPER techniques into the browser itself, without relying on the host.

The future of the web browser, HTML, and scripting languages will see many changes. These proposed changes to the runtime system and the browser will help make JavaScript easier to use securely, but they require a wide adoption of new technologies. GATEKEEPER provides a way for hosting sites to identify dangerous JavaScript widgets today.

6.4 Typing in JavaScript

A more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [6] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety. Other work has been done to devise a static type system that describes the JavaScript language [3, 4, 24]. These works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subests of JavaScript. As far as we can tell, none of these approaches have been applied to realistic bodies of code. GATEKEEPER uses a pointer analysis to reason about the JavaScript program in contrast to the type systems and analyses of these works. We feel that the ability to reason about pointers and the program call graph allows us to express more interesting security policies than we would be able otherwise.

7 Conclusions

We feel that static analysis of JavaScript is a key building block for enabling an environment in which code from different parties can safely co-exist and interact. This paper presents GATEKEEPER, a mostly static sound policy enforcement tool

for JavaScript. GATEKEEPER is built on top of what is to our knowledge the first pointer analysis developed for JavaScript. To show the practicality of our approach, we describe nine representative security and reliability policies for JavaScript widgets. Statically checking these policies results in 1,341 verified warnings in 684 widgets, with 113 false positives affecting only two widgets. While in this paper our focus is on policy enforcement, the techniques outlines here are generally useful for any task that involves reasoning about code such as code optimization, rewriting, program understanding tools, bug finding tools, etc.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] Ajaxian. Facebook JavaScript and security. `http://ajaxian.com/archives/facebook-javascript-and-security`, Aug. 2007.

[3] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS. Elsevier, 2004. http://www.binarylord.com/work/js0wood.pdf*, 2004.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *In ECOOP05 - Object-Oriented Programming, Lecture Notes in Computer Science*, pages 429–452. Springer, 2005.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85, 2006.

[6] R. Cartwright and M. Fagan. Soft typing. *SIGPLAN Not.*, 39(4):412–428, 2004.

[7] B. Chess, Y. T. O'Neil, and J. West. JavaScript hijacking. `www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, Mar. 2007.

[8] D. Crockford. Globals are evil. `http://yuiblog.com/blog/2006/06/01/global-domination/`, June 2006.

[9] D. Crockford. AdSafe: Making JavaScript safe for advertising. `http://www.adsafe.org`, 2007.

[10] D. Crockford. *JavaScript: the good parts*. 2008.

[11] Facebook, Inc. Fbjs. http://wiki.developers.facebook.com/index.php/FBJS, 2007.

[12] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proceedings of the Workshop on Social Network Systems*, pages 25–30, 2008.

[13] Finjan Inc. Web security trends report. http://www.finjan.com/GetObject.aspx?ObjId=506.

[14] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[15] javascript-reference.info. JavaScript obfuscators review. http://javascript-reference.info/javascript-obfuscators-review.htm, 2008.

[16] E. Kiciman and H. J. Wang. Live monitoring: using adaptive instrumentation and analysis to debug and maintain Web applications, *in submission*. 2007.

[17] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. Technical report, Microsoft Research, Feb. 2009.

[18] Microsoft Corporation. Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/SDV.mspx, 2005.

[19] Microsoft Live Labs. Live Labs Websandbox. http://websandbox.org, 2008.

[20] Microsoft Live Labs. Quality of service (QoS) protections. http://websandbox.livelabs.com/documentation/use_qos.aspx, 2008.

[21] Microsoft Security Bulletin. Vulnerabilities in Windows gadgets could allow remote code execution (938123). http://www.microsoft.com/technet/security/Bulletin/MS07-048.mspx, 2007.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. http://google-caja.googlecode.com/files/caja-2007.pdf, 2007.

[23] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI*, 2006.

[24] P. Thiemann. Towards a type system for analyzing JavaScript programs. 2005.

[25] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[26] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[27] Windows Live. Windows live gadget developer checklist. `http://dev.live.com/gadgets/sdk/docs/checklist.htm`, 2008.

[28] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of Conference on Principles of Programming Languages*, Jan. 2007.