Staged Information Flow for JavaScript *

Ravi Chugh Jeffrey A. Meister Ranjit Jhala Sorin Lerner

University of California, San Diego

{rchugh,jmeister,jhala,lerner}@cs.ucsd.edu

Abstract

Modern websites are powered by JavaScript, a flexible dynamic scripting language that executes in client browsers. A common paradigm in such websites is to include third-party JavaScript code in the form of libraries or advertisements. If this code were malicious, it could read sensitive information from the page or write to the location bar, thus redirecting the user to a malicious page, from which the entire machine could be compromised. We present an information-flow based approach for inferring the effects that a piece of JavaScript has on the website in order to ensure that key security properties are not violated. To handle dynamically loaded and generated JavaScript, we propose a framework for staging information flow properties. Our framework propagates information flow through the currently known code in order to compute a minimal set of syntactic residual checks that are performed on the remaining code when it is dynamically loaded. We have implemented a prototype framework for staging information flow. We describe our techniques for handling some difficult features of JavaScript and evaluate our system's performance on a variety of large real-world websites. Our experiments show that static information flow is feasible and efficient for JavaScript, and that our technique allows the enforcement of information-flow policies with almost no run-time overhead.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Validation; F.3.2 [*Semantics of Programming Languages*]: Semantics of Programming Languages – Program analysis

General Terms Languages, Reliability, Verification

Keywords Set Constraints, Flow Analysis, Web Applications, Confidentiality, Integrity

1. Introduction

JavaScript is a popular scripting language that is the foundation of Web 2.0 applications like Gmail and Facebook. The popularity of JavaScript stems from its extremely dynamic nature: libraries can be downloaded at run time from diverse sources across the web,

* This work was supported by NSF CAREER grants CCF-0644306, CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603.

PLDI'09, June 15–20, 2009, Dublin, Ireland.


```
<script src="http://adnetwork.com/insert-ad.js">

<textbox id="SearchBox">
<button id="Search" onclick="doSearch()">

<script type="javascript">
var doSearch = function() {
   var searchBox = document.nodes.SearchBox.value;
   var searchStr = searchUrl + searchBox;
   document.location.set(searchStr);
}
</script>
```

Figure 1. A snippet of JavaScript based on `www.wsj.com`. When the user clicks the Search button, the `doSearch` function appends the contents of the `SearchBox` to a base URL string `searchUrl`, and redirects the page to the resulting URL.

objects and code can be sent over the network as raw strings that are dynamically parsed and executed by the receiver, and all modern web browsers provide JavaScript APIs that allow scripts executing on the page to dynamically access and modify the state associated with the page. Unfortunately, the flexibility comes at a great price: JavaScript has few protection or information hiding mechanisms, and consequently, the use of JavaScript has opened up new classes of security vulnerabilities such as cross-site scripting and code-injection attacks.

We illustrate the main issues here with a simple and glaring attack found in a study of real-world vulnerabilities carried out at Google [27]. Figure 1 shows a code snippet adapted from `www.wsj.com`. On the first line, the web site inserts an ad by including some JavaScript code from an ad agency. This JavaScript code runs and replaces itself on the web page with the actual ad (this is a very common way of placing ads on web pages, including Google's AdSense). Below the ad, the sample page contains a search form with a text box and a search button with an on-click event handler. The event handler redirects to the search site stored in a global `searchUrl` variable (which can be reassigned), with the contents of the search box appended as a URL parameter.

In practice, first tier ad agencies often delegate to second tier agencies, which often delegate to third tier agencies, and so on. In their study, Google found a case where ads from a reputable and non-malicious American ad agency, after several levels of indirection, eventually included JavaScript code from a malicious ad provider in Russia. In the `wsj` example, such a malicious JavaScript snippet could simply write to `searchUrl`, redirecting the user to the attacker's site the next time the search button is clicked. This malicious site could then exploit a vulnerability in the browser to compromise the client's machine. Thus, the attacker can divert the user to the malicious site, *without directly changing* the document's location. The Google study reports that almost all web attacks which take over a user's computer follow this pattern: use

JavaScript to change the location bar to redirect to a malicious site that then exploits a vulnerability in the browser.

Although a browser's vulnerability is the last nail in the coffin, the root cause of the problem is that the code that is included for inserting the ad should not be able to change the location bar, and the designers of the `wsj` page certainly never intended to give the ad code this privilege. Thus, in order to make Web 2.0 applications secure, the fundamental challenge is to devise a mechanism that can specify and enforce the designer's intentions about the effects that a piece of JavaScript code can have.

In this paper, we propose to formalize these effects using *information flow*. Information flow can capture the fact that a particular value in the program *affects* another value in the program. In the above example with the location bar, the *integrity* policy that we would want to enforce is that no value from within any ad should flow to the location bar, or into any child frame's location bar. Similarly, using information flow, we could specify useful *confidentiality* properties like the sensitive cookie value of the current page should not flow to any ad, or information filled in textboxes on the current page should not flow to any third-party widgets (such as counters inserted on the page).

Although information flow has been explored in many settings, the dynamic nature of JavaScript poses a new challenge: because JavaScript commonly evaluates complex code strings that are built at run time or read from the network, the entire code is not available until the JavaScript program is already running. As a result, techniques for statically checking information flow are not directly applicable. One solution to this challenge would be to check information flow dynamically, but unfortunately this approach has some significant drawbacks. In addition to adding a possibly large run time overhead, it would also prevent developers from catching policy errors early on in the development process.

To address the dynamic nature of JavaScript, we propose in this paper a framework for *staging* integrity and confidentiality information flow properties. Staging consists of statically computing as much of the information flow as possible based on the known code, and leaving the remainder of the computation until more code becomes available. Since the residual checking must be performed within the browser, and must be performed every time new code is dynamically loaded, we must ensure that the residual checks can be performed efficiently.

In our staging framework, the heavyweight flow analysis is carried out just once on the server and its results are distilled into succinct *residual checks*, that enjoy two properties. First, they soundly describe the properties that are left to be checked on the remaining code once it becomes known; if each piece of loaded code passes the residual checks, the top level flow policy is guaranteed to hold. Second, they obviate the need for flow analysis within the browser; they are syntactically enforceable and can be efficiently discharged when the dynamically loaded code is parsed inside the browser. Thus, by performing the bulk of the analysis statically, staging allows the enforcement of information flow policies for dynamic web applications with almost no run-time overhead. To sum up, we make the following contributions in this paper.

- We present a framework for staging integrity and confidentiality information flow properties in JavaScript programs. Section 2 gives an overview of our framework using an illustrative example, and Section 3 describes the framework in more detail.
- We present an instantiation of the framework using set inclusion constraints. Section 4 describes how we use set constraints to capture direct and indirect flows through difficult-to-analyze features of JavaScript like dynamically created objects, fields, first-class functions, and prototypes. Our constraint-based flow analysis of JavaScript code is a contribution by itself, independent of the staging framework.
- We evaluate our analysis techniques and our staging framework on a variety of real-world web sites. In particular, in Section 5 we demonstrate the feasibility of staged information flow by showing that: (1) our approach scales to the Alexa top 100 [1] web sites, (2) the residual checks are orders of magnitude faster than checking the entire program and fast enough to run inside the client browser, and (3) our analysis is precise enough to correctly identify flows with a small false positive rate.

```
<script type="javascript">
var document.settings = {
   setBaseUrl = function(s) { this.baseUrl = s },
   setVersion = function(i) { this.version = i }
}

var initSettings = function(s, i) {
  document.settings.setBaseUrl(s);
  document.settings.setVersion(i);
}

initSettings("mysite.com/login.php", 1.0);

var login = function() {
  var pwd = document.nodes.PasswordTextBox.value;
  if (readCookie("doCheck") && pwd.length < 8) {
    document.alert("Password is too short!");
  } else {
    var user = document.nodes.UsernameTextBox.value;
    var params = "u=" + user + "&p=" + pwd;
    post(document.settings.baseUrl, params);
  }
}
</script>

<text id="UsernameTextBox"> <text id="PasswordTextBox">
<button id="ButtonLogin" onclick="login()">

<div id="AdNode">
  <script src="adserver.com/display.js">
</div>
```

Figure 2. `mysite.com` with username and password textboxes to allow logging in. The global function `initSettings` is intended to be called once to initialize settings used by the page. The page also loads a third-party script, which will be `eval`ed when the string is received from the network.

```
var z1 = "evil.com"; var z2 = 1.0; initSettings(z1,z2);
```

Figure 3. A *bad* network string `display.js`, returned by the malicious or compromised `adserver.com`, which calls `initSettings` to overwrite the page's settings. When the user clicks the login button, her username and password are sent to `evil.com` instead of `mysite.com`.

2. Overview

We start with an example that motivates our approach of staged information flow for JavaScript. Consider the web page shown in Figure 2. In our approach, we separate a web page into two parts. The first part, which we call the *context*, is known and in our example consists of the entire web page except for the last three lines. The second part, which we call the *hole*, is loaded dynamically and is unknown. In our example, the hole is the last

```
var displayAd;
var appName = document.navigator.appName;
var appVersion = document.navigator.appVersion;
if (appName == "IE" && appVersion < 7) {
  displayAd = function() { ... };
} else
  displayAd = function() { ... };
}
document.nodes.AdNode.innerHTML = displayAd();
```

Figure 4. A *good* network string `display.js`, which creates advertisement text depending on the browser detected from the `document.navigator` fields, and then displays it in the `AdNode` div that the page created for the ad.

three lines of the web page and consists of dynamically loaded third-party code from `adserver.com`.

The Context. The context contains some JavaScript code within `<script>...</script>` tags. This script defines a global object called `settings` that contains fields `baseUrl` and `version`, and associated setter methods `setBaseUrl` and `setVersion`. The function `initSettings` calls the setter methods to initialize the fields of the `settings` object. The function `login` reads the password from the appropriate field of the document and the document's cookie, creates a string comprised of the user's name and password, and calls `post` to send this string to the server whose URL is stored in `settings.baseUrl`. The `login` function is called asynchronously, on the reception of the event corresponding to the user clicking the `ButtonLogin` button. At the point when a user clicks on the `ButtonLogin` button, the `settings` object is already initialized with `mysite.com`, the intended destination of the username and password.

The Hole. Suppose that `adserver.com` is either malicious or compromised, and sends the JavaScript code shown in Figure 3. This code calls `initSettings` with values that cause the `baseUrl` field to point to the attacker's address. Thus, if the user presses the button *after* this dynamically added code is `eval`ed, the user's name and password will get posted to `evil.com` instead of `mysite.com`.

2.1 Safety via Information Flow

Unfortunately, existing mechanisms are insufficient to prevent this kind of attack. First, dynamic techniques like stack-based access control are insufficient in the face of asynchrony and global state. After the malicious code has set the global `settings.baseUrl`, it is no longer on the stack when the click event occurs, and hence there is nothing on the callstack to suggest that an attack has occurred. Second, JavaScript lacks most language-based, coarse-grained, information-hiding mechanisms such as private fields and abstract datatypes. This is a large part of JavaScript's appeal for Web 2.0 programming – by eschewing such mechanisms, JavaScript makes it easier to rapidly construct applications by gluing together trusted and untrusted libraries. Thus, to reconcile safety with flexibility, we need a fine-grained enforcement mechanism that allows untrusted code access to certain parts of the webpage, but ensures that critical elements cannot be affected.

Flow Policies. In our approach, the author of the context provides a flow *policy* which is a set of pairs of policy *elements*. A policy element is a program variable or a hole. Each pair in the policy represents flow that is *disallowed*. For our example, the attack above is possible because the code from the untrusted hole is able to interfere with a trusted part of the system, namely the URL to which the messages are sent. To shield the application from such attacks, the page's author could provide a policy that prohibits *any* variable declared within the hole from affecting the first parameter of the function `post`. Formally, we prevent untrusted `eval` sites from navigating the page via an *integrity* policy specified as a pair

$$(\bullet, \texttt{document.location})$$

which states that variables declared within the code loaded at *any* `eval` site *must not* flow into (*i.e.,* affect) the value of `document.location`. Dually, we can prevent secure information from being read by untrusted holes via a *confidentiality* policy specified as a pair

$$(\texttt{document.cookie}, \bullet)$$

which states that the value of `document.cookie` *must not* flow into any variable within the code loaded at *any* `eval` site.

Given an information flow policy and a complete program, there are standard techniques (*e.g.,* using type systems [23] or dataflow analysis [21]), for checking that the program satisfies the policy. Unfortunately, these techniques cannot be applied in our setting because of the extremely dynamic nature of JavaScript. At any time, additional code can be downloaded or dynamically generated using `eval`, which executes an arbitrary string as code. One option is to resort to fully dynamic enforcement of the flow policies. A second option is to re-analyze the entire application every time a new piece of code is loaded or `eval`ed. However, both of these options incur a significant runtime overhead, which may make certain applications unusable.

2.2 Staged Information Flow

Instead, our approach is to *stage* the analysis by pre-analyzing the code from the context using the given policy, in order to compute a *residual policy*, which captures the requirements that the hole must satisfy in order for the entire program to satisfy the flow policy. If the hole is filled with dynamically loaded code that contains more holes, then the staging framework recursively checks the residual policy on the inner holes, and so on.

Stage 1: Computing Residual Policy. Let us see how, for the example from Figure 2, the context's code can be used to reduce the flow policy that no variable from the hole should flow into the first parameter of `post`, into a residual policy that must be satisfied by the code loaded from `adserver.com`. To do so we use a static constraint-based analysis to compute the set of values that can flow into all known variables that are in the same scope as the hole and that can affect the first parameter of `post`. These include `document.settings.baseUrl`, the `s` parameter of `document.settings.setBaseUrl`, and the `s` parameter of `initSettings`. Thus, the residual policy states that no variable declared in the hole should flow into any of the above variables (in addition to the first formal of `post`).

Stage 2: Checking Residual Policy. Once the code is loaded from `adserver.com` we can rapidly check it against the residual policy. The code from Figure 3 violates the residual policy as `z1`, declared in the hole, is passed in as the first parameter for `initSettings`. Dually, if the code satisfies the residual policy (*i.e.,* if variables declared in the hole do not flow into variables known to affect the first parameter of `post`), then we are guaranteed that the original information policy holds over the complete system. For example, suppose that the code that gets loaded for the hole is that shown in Figure 4. In this case, the hole does not violate the residual policy as it only reads fields of the `document.navigator` object, and hence, we are guaranteed that the page is safe with respect to the given policy. Thus, the residual policy allows us to quickly check the hole when it is dynamically filled with code, without the check being slowed by tracking flows that occur within the context.

Our staged information flow framework can be used to split the analysis burden across client and server. Along with the development of a JavaScript application, the developer will specify the set of information flow policies that should be enforced on the page. During the testing cycle, various configurations and controlled libraries can be tested against the policy. The remaining residual checks can then be sent to the client along with the page, and a modified browser can perform the remaining stages of the analysis, halting execution if the policy is ever violated. We assume that the residual policies are not tampered with either when transmitted over the network or within the client browser.

In this setting, residual checks are performed by the client browser, which means that their efficiency directly affects the browsing experience. Consequently, to make residual checks as fast and simple as possible, we have designed our staging framework so that all residual checks are entirely *syntactic*, eliminating the need to do full-scale information flow on the client side.

2.3 Challenges of Analyzing JavaScript

JavaScript poses several challenges in addition to dynamic code loading and generation.

Functions. First, functions are objects, and hence first-class values that can be bound to other variables and passed around as parameters. For example, all the functions and methods in Figure 2 are created as anonymous function objects that are bound to the variables whose names are subsequently used to invoke the functions. Further, JavaScript programs make heavy use of first-class functions for several reasons, including to attach listeners to events, and to allow different versions of the same function to be defined based on run time properties, as shown in the case of the `displayAd` function in Figure 4. Thus, any analysis for JavaScript must be able to handle function values – which rules out the use of the standard summary-based interprocedural analyses that are used for first-order languages.

Fields. Second, most variables refer to objects that contain fields. Unlike statically typed languages like Java, it is difficult to determine which fields an object has because there are no classes, and fields can be dynamically added to objects. In essence, objects correspond to dictionaries and fields are simply used as names to look up in the dictionary. For instance, in our example, in order to be sufficiently precise, the analysis must be able to distinguish between the different fields of the `document` object. Thus, any analysis for JavaScript must be field-sensitive, and not lump together values flowing into different fields of an object. However, the sensitivity must be achieved efficiently as each object can have many fields, thereby making a naïve analysis impossible to scale.

Prototypes. Third, JavaScript eschews classes in favor of a form of inheritance called *prototyping*. In essence, each function `Foo` can be used as a constructor to create objects: the expression `new Foo(...)` creates an object and calls the function `Foo` to initialize the object. This way of constructing objects also creates an implicit inheritance chain through the use of a special field called `proto`. In particular, each function `Foo` has a corresponding *prototype object* that is implicitly constructed and stored in the field `Foo.proto` – recall that functions are objects and can therefore have fields.[1] When a new object `o` is constructed with the expression `new Foo(...)`, the new object implicitly inherits all the attributes of the `proto` field of `Foo` (that is, the new object essentially inherits from `Foo.proto`). This means that on each field read `o.f`, if `f` is not a field of `o`, then the prototype of the function that created `o` (in this case, `Foo`'s prototype) is used to lookup field `f`. Since `Foo`'s prototype is itself an object, it may also have been created using a function that had a prototype field. Thus, JavaScript transitively follows the chain of `proto` fields until the field being looked up is resolved, or until the root of the chain is reached without finding the field. Because the `proto` field can be read and written at any time, just like any other field, a JavaScript analysis must carefully track the prototype objects corresponding to each constructor function, and must track for each object, the attributes that the object implicitly inherits via the prototype chain.

[1] We use `proto` throughout as shorthand for `prototype`, which is the actual field name used in JavaScript. Also, each prototype object has a `constructor` field that holds the function value itself. Although we model this in our implementation, we omit the details for clarity of exposition.

e	$::=$		*Expressions:*
	$\mid$	$\mathtt{c}$	constant
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$\mathtt{x.f}$	field-read
	$\mid$	$e_1 \ \mathtt{op}\ e_2$	bin-op
	$\mid$	$\{\ldots, \mathtt{f}\!: e, \ldots\}_i$	object literal
	$\mid$	$\mathtt{this}$	this
	$\mid$	$\mathtt{fun}_i(\mathtt{this}_i, \mathtt{p}_i)\{\ s\ \}$	fun-def
	$\mid$	$\mathtt{f}(e)$	fun-call
	$\mid$	$\mathtt{y.f}(e)$	method-call
	$\mid$	$\mathtt{new}_i\ \mathtt{f}(e)$	constructor-call
s	$::=$		*Statements:*
	$\mid$	$\mathtt{skip}$	skip
	$\mid$	$\mathtt{var\ x}$	var-def
	$\mid$	$\mathtt{x} := e$	assign
	$\mid$	$\mathtt{x.f} := e$	field-assign
	$\mid$	$s_1; s_2$	sequence
	$\mid$	$\mathtt{if}_i\ \mathtt{x}\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2$	branch
	$\mid$	$\mathtt{while}_i\ \mathtt{x}\ \mathtt{do}\ s$	while
	$\mid$	$\mathtt{return}\ e$	return
	$\mid$	$\mathtt{eval}_i(\mathtt{x})$	eval
P	$::=$	$2^{(X \times \bullet) \cup (\bullet \times X)}$	*Policies*
RP	$::=$	$2^X \times 2^X$	*Res. Policies*

Figure 5. Syntax

3. Framework

We now describe our framework by formalizing the language, the notions of flow and residual policies, and finally describing how we stage the information flow analysis.

3.1 Core JavaScript

Figure 5 summarizes the syntax of Core JavaScript, which captures the essence of JavaScript.

Expressions of Core JavaScript include: basic constants `c` which include integers, $0, 1, \ldots$, strings, *etc.*; variable reads `x`; field reads `x.f`, where `f` is a field name; binary operations e_1 `op` e_2, where `op` includes primitive operations like addition, string concatenation *etc.*; function declarations, where each function is labeled by a unique identifier i, and has *two* formal parameters $\mathtt{this}_i$ and $\mathtt{p}_i$; function, method, and constructor calls, where exactly *one* parameter is passed to the callee; objects, which are a sequence of field-expression bindings; and the `this` expression.

Statements of Core JavaScript include: variable declarations, variable and field assignments, statement sequencing, branching and while-loops, and return statements. To model dynamic code loading, we include an `eval` statement. A statement is *open* if it contains an `eval` site, and *closed* otherwise.

Conventions. We assume, without loss of generality, that the program satisfies certain syntactically enforceable conventions. First, we assume that all functions are declared anonymously, and that each declaration has a unique label i. Further, each function has exactly two parameters. The first parameter $\mathtt{this}_i$ corresponds to the object that will be referred to as `this` inside the body of the function. We use $\texttt{this}_i$ to model JavaScript's semantics for `this`. The second parameter $\texttt{p}_i$ corresponds to the argument passed to the function. When a function is called *directly* as $\texttt{f}(e)$, the `this` variable is set to the *global object*, and the value that e evaluates to is passed in as the second parameter. When a function is called *indirectly* as a method call $\texttt{x.f}(e)$, the object `x` is passed in as the first parameter, and the value that e evaluates to is passed as the second parameter. Second, we assume that each `eval` statement is uniquely labeled. Further, we assume that each branch and loop is uniquely labeled – we will use these labels to compute indirect flows, which are the flows that occur from the branch value to locations being assigned in the branch. Third, we assume that each variable is declared, and that all variables are renamed so that local variables in different scopes have unique names. Fourth, rather than explicitly modeling the webpage (*i.e.,* encoding the DOM), we assume that there is an object `document` in the global namespace that can be accessed and manipulated via the appropriate fields and methods.

Dynamic Semantics. The dynamic semantics of Core JavaScript are standard – we refer the reader to [31, 36, 6, 17] for a detailed formalization via small-step operational semantics.

3.2 Information Flow Policies

Our information flow policies are expressed as sets of pairs, where each pair represents a must-not-flow requirement. In its most general form, such policies would include pairs where each element is either a variable or the label of an `eval` site. Although our unstaged information flow analysis will handle such general flow policies, these policies make residual checks difficult to perform syntactically, and require sending a large amount of state across stages. Consider for example a policy stating that a variable `x` should not flow to another variable `y`, and a context containing a single assignment `b = a`. If the hole executes `a = x` and `b = y`, then the flow policy is violated. Since the hole can add flow between any variables `a` and `b` that it chooses, in the most general case the first stage must send to the second stage all possible flows of variables in the context. This would require sending a large amount of data to the second stage, and would also require performing a full flow analysis in the client browser.

Thus, to make our policies more amenable to syntactic residual checks, we restrict ourselves to confidentiality and integrity policies: confidentiality policies state that sensitive information should not be leaked, whereas integrity policies state that the attacker cannot compromise sensitive information. As a result, these policies include pairs where one element of the pair is a variable and the other is a hole, which would not allow the problematic policy $(\texttt{x}, \texttt{y})$ to be expressed.

Even with this restriction, policies that grant one hole access to a sensitive variable but not another pose problems for staging. In particular, consider the case where the client receives a hole that is allowed to access the sensitive variable. This hole may induce flow that must be taken into account in the residual checks for the remaining holes that cannot access the variable. To update the residual policy would again require a complex computation on the client. As a result, instead of allowing must-not-flow pairs to be specific to particular holes, we require that if the policy restricts access to one hole, it restricts access to all holes.

Policies. Formally, we define a must-not-flow policy as a pair of the form $(\texttt{x}, \bullet)$, which states the *confidentiality* policy that the value of the variable `x` *must not* flow to any variable within a hole, or $(\bullet, \texttt{x})$, which states the *integrity* policy that values from variables within a hole *must not* flow into `x`. A *policy* P is a set of must-not-flow policies. A *residual policy* RP is a pair of two sets of variables or fields MNR and MNW called the a *must not read* set and *must not write* set respectively. Intuitively, the code loaded at any hole must not read (resp. write) any variable or field in MNR (resp. MNW).

$$
\begin{array}{l}
\mathsf{SIF}(P, s) = \\
\quad RP \leftarrow \mathsf{Stage}(P, s) \\
\quad c \leftarrow \mathsf{Initialize}(s) \\
\quad \textbf{do} \\
\qquad (s, c) \leftarrow \mathsf{Execute}(c) \\
\quad \textbf{while } (c \neq \textsc{Exit} \text{ and } \mathsf{Check}(RP, s) \neq \textsc{Err}) \\
\quad \textbf{if } c = \textsc{Exit} \textbf{ then return } \textsc{Safe} \textbf{ else return } \textsc{Err}
\end{array}
$$

Figure 6. Staged Information Flow Framework

Even though policies are restricted to variables, notice that we can stipulate a policy like $(\texttt{x.f}, \bullet)$ (resp. $(\bullet, \texttt{y.g})$) by (a) creating a new variable $\texttt{x}'$ (resp. $\texttt{y}'$), (b) adding a new assignment $\texttt{x}' := \texttt{x.f}$ (resp. $\texttt{y.g} := \texttt{y}'$) and, (c) specifying a policy $(\texttt{x}', \bullet)$ (resp. $(\bullet, \texttt{y}')$). Similarly, to prevent flows from constants we assume that dynamically loaded code is rewritten so that all constants that appear in the hole are bound to new variables declared within the hole.

Flows. Informally, we say that a variable `x` *flows into* `y` if the value of `y` can be affected by the value of `x`, either *directly*, via a sequence of assignments, or *indirectly*, due to conditional dependences. To formalize when a flow policy P is violated by a program s we rewrite the program to $\mathsf{Rewrite}(P, s)$, which is an instrumented version of s that: (1) has auxiliary taint fields that track flows, and (2) calls a special Core JavaScript function `flowDetected()` as soon as a flow is detected during execution from `x` to a hole, for some $(\texttt{x}, \bullet) \in P$ or from a hole to `x`, for some $(\bullet, \texttt{x}) \in P$. Figure 7 formalizes the most important cases of the rewriting function Rewrite. Notice that the rewriting function is recursively invoked every time an `eval` is executed.

Flow Policy Satisfaction. We say that a program s *violates* a policy P if there is an execution of $\mathsf{Rewrite}(P, s)$ along which the function `flowDetected()` is called. Otherwise, we say that s *satisfies* the policy P.

3.3 Staged Policy Verification

Procedure SIF. Figure 6 formalizes our Staged Information Flow framework as a procedure SIF that takes as input a policy P and an *open* Core JavaScript program s and returns either SAFE indicating that the policy P was satisfied or ERR indicating that the policy was violated. At a high level, our framework stages information flow checking as follows. First, the framework calls Stage with the top-level policy P and the context s, *i.e.,* the known part of the program, to compute the flows that occur in the context. Stage uses the computed flows and the (top-level) policy to pre-compute and return a *residual* policy RP that is a projection of the (top-level) policy to the `eval` sites. Second, the framework initializes the variable c with a snapshot of the entire initial state of the executing program. Third, the framework enters a loop where it invokes Execute on the snapshot c to run the program until it reaches the next `eval` site, or terminates. In the former case, Execute returns a pair (s, c) where s is the code to be loaded at the next site, and c is the current snapshot of the program. In the latter case, Execute returns a triple where c is simply EXIT, indicating the program has terminated. The loop is repeated until the program terminates or Check determines that the loaded code s violates the residual policy RP. After breaking out of the loop, we check if we exited because the program terminated or, because of some call to Check, returned ERR. In the former case SIF returns SAFE, and in the latter ERR.

Procedures Stage and Check. The framework is parameterized by two procedures Stage and Check. Stage takes as input a policy P and a statement s corresponding to a context, and returns a residual policy RP corresponding to the projection of P to the `eval` sites in s. Check takes as input a residual policy RP and a statement s corresponding to code loaded in at a hole, and returns ERR or SAFE.

Soundness. For two programs s, s' and hole label i, let $s[i \mapsto s']$ be the closed program obtained by replacing the `eval` site i in s with s' and all other `eval` sites with `skip`. To ensure soundness, the procedures Stage and Check must meet the following requirements which state that the procedures must *overapproximate* the flows that occur in concrete executions.

$$\forall P, s, i, s'.\ \text{if } s[i \mapsto s'] \text{ violates } P \quad \text{then } \mathsf{Check}(\mathsf{Stage}(P, s), s') = \text{ERR}$$

$$\forall P, s, i, s'.\ \text{if } \mathsf{Check}(\mathsf{Stage}(P, s), s') = \text{SAFE} \quad \text{then } \mathsf{Stage}(P, s) = \mathsf{Stage}(P, s[i \mapsto s'])$$

We can show that if Check and Stage meet the above criteria, then for all policies P and programs s, if P is violated by s then $\mathsf{SIF}(P, s)$ returns ERR. A sketch of the proof is as follows: assume that P is violated at some point during the execution of s, and suppose that at the point of failure, sites i_1 through i_k in s had been loaded with s_1 through s_k. Thus, we know that $s[i_1 \mapsto s_1, \cdots, i_k \mapsto s_k]$ violates the policy, and then by the first property above, we know that $\mathsf{Check}(\mathsf{Stage}(P, s[i_1 \mapsto s_1, \cdots, i_{k-1} \mapsto s_{k-1}]), s_k) = \text{ERR}$. Then, using $k-2$ applications of the second property above, we can show that $\mathsf{Stage}(P, s[i_1 \mapsto s_1, \cdots, i_{k-1} \mapsto s_{k-1}]) = \mathsf{Stage}(P, s)$, and therefore $\mathsf{Check}(\mathsf{Stage}(P, s), s_k) = \text{ERR}$, which means that $\mathsf{SIF}(P, s)$ would return ERR when it performs the residual check on s_k.

The second condition above enables our framework to compute the flows and residual policies once, without having to recompute them each time that a hole is filled. In essence, the conditions on Stage and Check ensure that the dynamically loaded code does not induce any new flows for the variables described in the top-level policy P. If any new flows would be induced by the hole, then Check would return ERR and execution would be halted.

4. Static Instantiation

We now describe how we have instantiated our framework by presenting our implementations of Stage and Check. Stage takes a policy P and program s and returns the residual policy for the `eval` sites in s. Check takes a residual policy RP comprising a must-not-read and must-not-write set, and a statement corresponding to code to be loaded at a hole, and verifies that the statement satisfies the residual policy, by verifying that the statement does not read (resp. write) the variables or fields listed in the must not read (resp. must not write) sets.

Next, we describe our flow-insensitive, field-sensitive, set-constraint based instantiation of the procedure Stage. First, we present the different elements constituting the constraints, constants, constructors, and terms. Second, we describe our syntax-directed constraint generation procedure. Third, we discuss some optimizations required to analyze JavaScript with sufficient precision. Fourth, we show how Stage combines policies and constraints to compute residual policies.

Set Constraints. A *term* is either a *constraint variable* $\mathcal{X}$, a *constant*, or a *constructed term* $\mathrm{C}(t_1, \ldots, t_n)$, where C is a *constructor* of arity n and $t_1, \ldots, t_n$ are terms. A *set constraint* is a constraint of the form $t_1 \subseteq t_2$, where t_1 and t_2 are terms. A satisfying solution for a finite set of constraints maps each constraint variable to a set of constants and constructed terms, such that all of the inclusion constraints are satisfied. For details, we refer the reader to [20]. For two unary constructors C, D, we write the constraint $t_1 \subseteq_{C,D} t_2$ as an abbreviation for the pair of constraints $t_1 \subseteq C(\mathcal{X})$, $D(\mathcal{X}) \subseteq t_2$, where $\mathcal{X}$ is a fresh constraint variable that is distinct from all other variables.

```
SRC(P, z) =
  if (z, •) ∈ P or
     z is a hole var
  then [z] else []

DST(P, z) =
  if ∃a s.t. a ∈ z.taint and
        [((a, •) ∈ P and z is a hole var ) or
        ((•, z) ∈ P and a is a hole var )]
  then flowDetected()

Rewrite(P, c) = {data: c, taint: I}
Rewrite(P, var x) = var x
Rewrite(P, return x) = return x

Rewrite(P, x) =
   {data: x.data,
    taint: I + x.taint + SRC(P, x)}

Rewrite(P, x.f) =
   {data: x.data.f.data,
    taint: I + x.data.f.taint}

Rewrite(P, x op y) =
   {data: x.data op y.data,
    taint: I + x.taint + y.taint}

Rewrite(P, fun(this, p){ s }) =
   {data: fun(this, x, I)Rewrite(P, s),
    taint: I}

Rewrite(P, {f1: x1, ...}) =
   {data: {f1: {data: x1.data, taint: I + x1.taint}, ...},
    taint: I}

Rewrite(P, x := e) =
   var tmp := Rewrite(P, e);
   x.data := tmp.data;
   x.taint := tmp.taint;
   DST(P, x)

Rewrite(P, x.f := e) =
   var tmp := Rewrite(P, e);
   x.data.f.data := tmp.data;
   x.data.f.taint := tmp.taint

Rewrite(P, x := f(z)) =
   var tmp := f.data;
   x := tmp(this, z, I + f.taint);
   DST(P, x)

Rewrite(P, x := y.f(z)) =
   x := y.data.f.data(y, z, I + y.data.f.taint);
   DST(P, x)

Rewrite(P, s1;s2) =
   Rewrite(P, s1);Rewrite(P, s2)

Rewrite(P, if_i x then s1 else s2) =
   var tmp := I;
   I := I + x.taint;
   if_i x.data then Rewrite(P, s1) else Rewrite(P, s2);
   I := tmp

Rewrite(P, while_x s do ) =
   var tmp := I;
   I := I + x.taint;
   while_x.data Rewrite(P, s) do ;
   I := tmp
Rewrite(P, x := eval_i(y))
   var tmp := eval_i(Rewrite(P, y.data));
   x.data := tmp.data;
   x.taint := I + tmp.taint + y.taint
```

Figure 7. Dynamic Information Flow Rewriting. We assume complex expressions are bound to fresh temporary variables. The global variable `I`, initially the empty set, stores the set of indirect taints.

Statements

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{skip}) = \emptyset$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{var\ x}) =$
$\quad \{\mathtt{c_x} \subseteq \mathcal{X}_\mathtt{x}\} \cup \{\mathrm{Ind}(\mathtt{c_x}) \subseteq \mathcal{X}_\mathtt{x}\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{x} := e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e) \cup \{\mathcal{X}_e \subseteq \mathcal{X}_\mathtt{x}\} \cup \{\mathcal{X}_I \subseteq \mathcal{X}_\mathtt{x}\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{x.f} := e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e) \cup$
$\quad \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Real}(\mathrm{Fld}_\mathtt{f}(\mathcal{X}_e, \Omega))\} \cup \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Real}(\mathrm{Fld}_\mathtt{f}(\mathcal{X}_I, \Omega))\}$

$\mathsf{Gen}(k, \mathcal{X}_I, s_1; s_2) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, s_1) \cup \mathsf{Gen}(k, \mathcal{X}_I, s_2)$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{if}_i\ \mathtt{x}\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_i, s_1) \cup \mathsf{Gen}(k, \mathcal{X}_i, s_2) \cup$
$\quad \{\mathcal{X}_I \subseteq \mathcal{X}_i\} \cup \{\mathcal{X}_\mathtt{x} \subseteq_{\mathrm{Ind,Ind}} \mathcal{X}_i\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{while}_i\ \mathtt{x}\ \mathtt{do}\ s\ ) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_i, s) \cup \{\mathcal{X}_I \subseteq \mathcal{X}_i\} \cup \{\mathcal{X}_\mathtt{x} \subseteq_{\mathrm{Ind,Ind}} \mathcal{X}_i\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{return}\ e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e) \cup \{\mathcal{X}_e \subseteq \mathcal{X}_{\mathtt{ret}_k}\} \cup \{\mathcal{X}_I \subseteq \mathcal{X}_{\mathtt{ret}_k}\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{eval}_i(e)) = \emptyset$

Expressions

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{c}\ \mathrm{as}\ e) =$
$\quad \{\mathtt{c} \subseteq \mathcal{X}_e\} \cup \{\mathrm{Ind}(\mathtt{c}) \subseteq \mathcal{X}_e\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{x}) = \emptyset$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{x.f}\ \mathrm{as}\ e) =$
$\quad \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Real}(\mathrm{Fld}_\mathtt{f}(\emptyset, \mathcal{X}_e))\} \cup \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Pro}(\mathrm{Fld}_\mathtt{f}(\emptyset, \mathcal{X}_e))\}$

$\mathsf{Gen}(k, \mathcal{X}_I, e_1\ \mathtt{op}\ e_2\ \mathrm{as}\ e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e_1) \cup \mathsf{Gen}(k, \mathcal{X}_I, e_2) \cup$
$\quad \{\mathcal{X}_{e_1} \subseteq \mathcal{X}_e\} \cup \{\mathcal{X}_{e_2} \subseteq \mathcal{X}_e\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \{\ldots, \mathtt{f}_j : e_j, \ldots\}_i\ \mathrm{as}\ e) =$
$\quad (\cup_j \mathsf{Gen}(k, \mathcal{X}_I, e_j)) \cup \{\mathcal{X}_i \subseteq \mathcal{X}_e\} \cup$
$\quad (\cup_j \{\mathcal{X}_{e_j} \subseteq \mathcal{X}_{i.\mathtt{f}_j}\}) \cup (\cup_j \{\mathcal{X}_I \subseteq \mathcal{X}_{i.\mathtt{f}_j}\}) \cup$
$\quad (\cup_j \{\mathrm{Real}(\mathrm{Fld}_{\mathtt{f}_j}(\mathcal{X}_{i.\mathtt{f}_j}, \mathcal{X}_{i.\mathtt{f}_j})) \subseteq \mathcal{X}_i\})$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{this}\ \mathrm{as}\ e) = \{\mathtt{this}_k \subseteq \mathcal{X}_e\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{fun}_i(\mathtt{this}_i, \mathtt{p}_i)\{\ s\ \}\ \mathrm{as}\ e) =$
$\quad \mathsf{Gen}(i, \mathcal{X}_{\mathtt{ind}_i}, s) \cup \{\mathcal{X}_I \subseteq \mathcal{X}_{\mathtt{ind}_i}\} \cup \{\mathcal{X}_i \subseteq \mathcal{X}_e\} \cup$
$\quad \{\mathrm{Fun}(\mathcal{X}_{\mathtt{cons}_i}, \mathcal{X}_{\mathtt{this}_i}, \mathcal{X}_{\mathtt{p}_i}, \mathcal{X}_{\mathtt{ind}_i}, \mathcal{X}_{\mathtt{ret}_i}) \subseteq \mathcal{X}_i\} \cup$
$\quad (\cup_j \{\mathrm{Real}(\mathrm{Fld}_{\mathtt{f}_j}(\mathcal{X}_{\mathtt{proto}_i.\mathtt{f}_j}, \mathcal{X}_{\mathtt{proto}_i.\mathtt{f}_j})) \subseteq \mathcal{X}_{\mathtt{proto}_i}\}) \cup$
$\quad \{\mathrm{Real}(\mathrm{Fld}_\mathtt{proto}(\mathcal{X}_{\mathtt{proto}_i}, \mathcal{X}_{\mathtt{proto}_i})) \subseteq \mathcal{X}_i\} \cup$
$\quad \{\mathcal{X}_{\mathtt{cons}_i} \subseteq \mathrm{Real}(\mathrm{Fld}_\mathtt{proto}(\mathcal{X}_{\mathtt{proto}_i}, \Omega))\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{f}(e')\ \mathrm{as}\ e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e') \cup \{\mathcal{X}_\mathtt{f} \subseteq \mathrm{Fun}(\emptyset, \mathcal{X}_{\mathtt{o}_g}, \mathcal{X}_{e'}, \mathcal{X}_I, \mathcal{X}_e)\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{x.f}(e')\ \mathrm{as}\ e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e') \cup$
$\quad \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Real}(\mathrm{Fld}_\mathtt{f}(\emptyset, \mathrm{Fun}(\emptyset, \mathcal{X}_\mathtt{x}, \mathcal{X}_{e'}, \mathcal{X}_I, \mathcal{X}_e)))\} \cup$
$\quad \{\mathcal{X}_\mathtt{x} \subseteq \mathrm{Pro}(\mathrm{Fld}_\mathtt{f}(\emptyset, \mathrm{Fun}(\emptyset, \mathcal{X}_\mathtt{x}, \mathcal{X}_{e'}, \mathcal{X}_I, \mathcal{X}_e)))\}$

$\mathsf{Gen}(k, \mathcal{X}_I, \mathtt{new}_i\ \mathtt{f}(e')\ \mathrm{as}\ e) =$
$\quad \mathsf{Gen}(k, \mathcal{X}_I, e') \cup \{\mathcal{X}_i \subseteq \mathcal{X}_e\} \cup$
$\quad (\cup_j \{\mathrm{Real}(\mathrm{Fld}_{\mathtt{f}_j}(\mathcal{X}_{i.\mathtt{f}_j}, \mathcal{X}_{i.\mathtt{f}_j})) \subseteq \mathcal{X}_i\}) \cup$
$\quad \{\mathcal{X}_\mathtt{f} \subseteq \mathrm{Fun}(\mathcal{X}_i, \mathcal{X}_i, \mathcal{X}_{e'}, \mathcal{X}_I, \Omega)\} \cup$
$\quad \{\mathcal{X}_{i.\mathtt{proto}} \subseteq_{\mathrm{Real,Pro}} \mathcal{X}_i\} \cup \{\mathcal{X}_{i.\mathtt{proto}} \subseteq_{\mathrm{Pro,Pro}} \mathcal{X}_i\}$

Figure 8. Constraint generation

4.1 Constraint Elements

We set up a system of constraints over variables $\mathcal{X}_e$ for each subexpression e of the program. The constraints use several kinds of constructors to model various aspects of JavaScript code. The first two constructors are standard ways of encoding functions and fields using set constraints [12]. The last three are novel mechanisms required to capture information flow and the semantics of JavaScript: they are used to distinguish between fields that are directly contained in an object, fields that are reachable by transitively following a *prototype* chain, values that *directly* reach a particular point, and values that *indirectly* reach a particular point.

1. Function Constructor. JavaScript programs have first class functions in that functions can be created and passed around like any other value. We model the flow of function values via a constructor $\mathrm{Fun}()$ of arity 5. The first argument corresponds to the objects *constructed* by the function object, a special feature of JavaScript that we will describe in the sequel. This argument is treated as *contravariant*. The second argument corresponds to the function's implicit parameter `this`. As the argument corresponds to an input of the function, it is treated as *contravariant*. The third argument corresponds to the explicit formal parameter of the function. As this argument also corresponds to an input of the function, it is treated as *contravariant*. The fourth argument corresponds to an implicit parameter that holds the values corresponding to indirect flows into the points where the function is invoked. This parameter is used as the initial set of indirect flows into the body of the function, and as it corresponds to an input, the argument is also treated is *contravariant*. The fifth argument corresponds to the return value, and hence the argument is *covariant*.

2. Field Constructors. JavaScript programs make heavy use of fields and any precise analysis must track flows in a field-sensitive manner. The classical way to model fields is to view them as a pair of functions: a *setter* that updates the contents of the field, and a *getter* the returns the contents of the field. Following this intuition, we encode a field `f` via a a constructor $\mathrm{Fld}_\mathtt{f}()$ of arity 2. The first parameter corresponds to the set of values written into the field, *i.e.,* the inputs to the setter, and hence, is treated as *contravariant*. The second parameter corresponds to the set of values read from the field, *i.e.,* the outputs of the getter, and hence, is treated as *covariant*. When initializing an object's fields, we use the same set variable in both places so that all arguments that flows into the first argument flow out of the second argument. When writing a field, we pad the second argument with the set variable Ω, which collects everything that flows from the field by covariance. When reading a field, we pad the first argument with the set variable $\emptyset$, so that nothing flows into the field by contravariance.

3. Real and Prototype Flow Constructors. In order to determine what a field read returns in the presence of prototyping, we must track, for each object, the values for all fields that can be read directly from the object or transitively via following its prototype chain. To distinguish between the fields of an object and the fields reachable via the prototype chain of an object, we use a special constructor $\mathrm{Real}()$ of arity 1 to wrap the fields that an object directly contains, and a special constructor $\mathrm{Pro}()$ of arity 1 to wrap the fields that are transitively reachable by following the object's prototype chain.

In general, if `c` can be reached by following the prototype chain of an expression e, then our constraints will ensure that $\mathrm{Pro}(\mathtt{c})$ flows into $\mathcal{X}_e$. For example, `x.f` can return the constant `c` if the object has a field `f` that has the value `c`, or if an object in its prototype chain has a field `f` that has the value `c`. In the former case, our constraints ensure that the term $\mathrm{Real}(\mathrm{Fld}_\mathtt{f}(\mathtt{c}, \mathtt{c}))$ flows into $\mathcal{X}_\mathtt{x}$. In the latter case, our constraints ensure that the term $\mathrm{Pro}(\mathrm{Fld}_\mathtt{f}(\mathtt{c}, \mathtt{c}))$ flows into $\mathcal{X}_\mathtt{x}$.

4. Indirect Flow Constructor. When tracking information flow, we must track both *direct* value flows, as well as *indirect* flows that arise when assignments take place under particular branch conditions. However, due to the presence of higher-order functions and dynamic dispatch, we must take care to separate direct flows (which affect which functions get executed at a different program points), from indirect flows (which have no effect on the execution). To achieve this separation, we use a covariant constructor $\mathrm{Ind}()$ of arity 1 to wrap constants and convert them into ground terms that participate in indirect flows.

In general, if a constant `c` directly flows into an expression e, then the constraints ensure that the term `c` flows into $\mathcal{X}_e$. If the constant `c` *indirectly* flows into an expression e, however, then our constraints ensure that $\mathrm{Ind}(\mathtt{c})$ flows into $\mathcal{X}_e$. For example, if `c` indirectly flows into the expression `x.f`, then our constraints will ensure that the term $\mathrm{Real}(\mathrm{Fld}_{\mathtt{f}}(\mathrm{Ind}(\mathtt{c}), \mathrm{Ind}(\mathtt{c})))$ flows into $\mathcal{X}_{\mathtt{x}}$.

4.2 Constraint Generation

Figure 8 shows the constraint generation procedure Gen. The procedure takes as input a label k corresponding to the identifier of the function currently being analyzed, a constraint variable $\mathcal{X}_I$ representing the indirect flows into the program location being analyzed, and either e or s, respectively the expression or statement being analyzed, and it returns as output a set of inclusion constraints. Gen traverses the AST of the program and generates constraints between variables of the form $\mathcal{X}_e$ for each subexpression e, that capture the set of values that flow directly or indirectly into e. We maintain the invariants that: (1) only values wrapped with the $\mathrm{Ind}()$ constructor flow into the indirect flow variables $\mathcal{X}_I$, (2) for every value that directly flows into e, there is a corresponding term that flows into $\mathcal{X}_e$, and, (3) for every value that is reachable from e after transitively following the prototype chain rooted at e, there is a corresponding term wrapped under $\mathrm{Pro}()$ that flows to $\mathcal{X}_e$. Next, we discuss how constraints are generated for a representative subset of expressions and statements.

Assignments. For each assignment $\mathtt{x} := e$, we generate constraints on the subexpression e, and then constraints that capture the direct flow from e into `x` as well as the indirect flow from the current location's indirect flow variable into `x`. Notice that a `return` statement is treated as an assignment to the return variable of the function to which the statement belongs.

Branches. Each branch statement and loop is labeled with a unique label i that can be generated with a syntactic pass over the source. For each branch or loop labeled i, we create a new indirect flow variable $\mathcal{X}_i$. We flow the values in $\mathcal{X}_I$ and the indirect values from the expression used in the branch condition into $\mathcal{X}_i$, and then use $\mathcal{X}_i$ as the indirect flow variable when generating the constraints for the statements that depend on the branch. To preserve the invariant that indirect flows are wrapped under the $\mathrm{Ind}()$ constructor (and hence, not used to affect computation), we *filter* flows to wrapped terms, using $\mathcal{X}_{\mathtt{x}} \subseteq_{\mathrm{Ind},\mathrm{Ind}} \mathcal{X}_i$.

Object Literals. We use the set variable $\mathcal{X}_{i.\mathtt{f}}$ to track the contents of field `f` of object i. For every field $\mathtt{f}_j$, we flow the initial value e_j into the field with the constraint $\mathcal{X}_{e_j} \subseteq \mathcal{X}_{i.\mathtt{f}_j}$, and the current indirect taint into the field with the constraint $\mathcal{X}_I \subseteq \mathcal{X}_{i.\mathtt{f}_j}$. Finally, we add the constraint $\mathrm{Real}(\mathrm{Fld}_{\mathtt{f}_j}(\mathcal{X}_{i.\mathtt{f}_j}, \mathcal{X}_{i.\mathtt{f}_j})) \subseteq \mathcal{X}_i$ to treat $\mathtt{f}_j$ as a field that object i directly contains, where $\mathcal{X}_{i.\mathtt{f}_j}$ is used as both the setter and getter for the field.

Fields. For each field `x.f` read in (resp. written in) an expression e, we create the appropriate flow constraint between $\mathcal{X}_{\mathtt{x}}$ and $\mathrm{Fld}_{\mathtt{f}}(\emptyset, \mathcal{X}_e)$ (resp. $\mathrm{Fld}_{\mathtt{f}}(\mathcal{X}_e, \Omega)$), which by virtue of the constructor matching and variance, has the effect of flowing the values from (resp. into) the `f` field of `x` into (resp. from) $\mathcal{X}_e$. We must account for prototype chains when reading and writing to fields. For field reads from `x.f`, the result can flow from either the object `x` (if it has a field `f`), or from some object in its prototype chain with a fields `f`. To model these semantics, the values returned from reads are the values from objects that directly flow into `x` (*i.e.,* $\mathrm{Real}()$-wrapped terms that flow into $\mathcal{X}_{\mathtt{x}}$) as well as objects that flow into `x` after following the prototype chain (*i.e.,* $\mathrm{Pro}()$-wrapped terms that flow into $\mathcal{X}_{\mathtt{x}}$). For fields writes to `x.f`, only the actual object itself may be updated (as opposed to some object along the prototype chain). To model these semantics, we only carry out the assignment on those objects that directly reach `x`, *i.e.,* are wrapped under $\mathrm{Real}()$.

Function Definitions. Each anonymous function declaration is labeled with a unique label i. For each function i, we create a term $\mathrm{Fun}(\mathcal{X}_{\mathtt{cons}_i}, \mathcal{X}_{\mathtt{this}_i}, \mathcal{X}_{\mathtt{p}_i}, \mathcal{X}_{\mathtt{ind}_i}, \mathcal{X}_{\mathtt{ret}_i})$ corresponding to the function's value, and create the appropriate constraints from the constraint variables representing the "inputs" $\mathcal{X}_{\mathtt{this}_i}$, $\mathcal{X}_{\mathtt{p}_i}$, and $\mathcal{X}_{\mathtt{ind}_i}$ into the body of the function, and from the return statement of the function to $\mathcal{X}_{\mathtt{ret}_i}$. The contravariant argument $\mathtt{cons}_i$ in the first position of the function term corresponds to the objects *constructed by* the function. The last three constraints deal with prototypes. First, we create a fresh prototype object, namely $\mathcal{X}_{\mathtt{proto}_i}$, and set up its fields in the same way we do for object literals. Second, we add the constraint $\mathrm{Real}(\mathrm{Fld}_{\mathtt{proto}}(\mathcal{X}_{\mathtt{proto}_i}, \mathcal{X}_{\mathtt{proto}_i})) \subseteq \mathcal{X}_i$ that stores the prototype object in the `proto` field of the function object. Third, we add the constraint $\mathcal{X}_{\mathtt{cons}_i} \subseteq \mathrm{Real}(\mathrm{Fld}_{\mathtt{proto}}(\mathcal{X}_{\mathtt{proto}_i}, \Omega))$, which has the effect of writing the prototype object into the `proto` field of any objects that flow to $\mathtt{cons}_i$.

Function Calls. For each function call, we generate a constraint that uses constructor *matching* to pull out the set of actual functions reaching the callsite, and uses variance to flow the actuals (both explicit, and implicit due to indirect flow) into the formals, and the return out to the callsite respectively [11]. The values flowed in for the `cons` and `this` parameter differ depending on the the three kinds of function calls.

- For *direct* calls of the form $\mathtt{f}(e')$, we use $\emptyset$ for the constructed object and the flow variable corresponding to the *global object* $\mathtt{o}_g$ for the `this` parameter.
- For *method* calls of the form $\mathtt{x.f}(e')$, we use constructor matching in a manner similar to field-reads, to pull out the appropriate functions that flow to the callsite, and we use the receiver object `x` as the `this` parameter.
- For *constructor* calls of the form $\mathtt{new}_i\ \mathtt{f}(e')$, which create a fresh object and call `f` with `this` bound to the fresh object, we introduce a new variable $\mathcal{X}_i$ that represents the objects created at the callsite and set up its fields as we do for object literals. Next, we use constructor matching to pull out the functions that flow to the callsite, and (via contravariance), flow $\mathcal{X}_i$ into the constructed object parameter and `this` parameters of the callees. Finally, the last two constraints "flatten" the constructed object's prototype chain. Intuitively, the constraints add all fields of an object's prototype chain into the object directly, while at the same time keeping track of which fields actually belong to the object and which do not. To achieve this, we take each object that flows into the prototype field of the constructed object ($\mathcal{X}_{i.\mathtt{proto}}$) either directly (*i.e.,* wrapped under $\mathrm{Real}()$) or via prototype-chains (*i.e.,* wrapped under $\mathrm{Pro}()$), and rewrap those objects under $\mathrm{Pro}()$ and flow them into the constructed object $\mathcal{X}_i$.

4.3 Analyzing Real JavaScript

Multi-parameter Functions. Functions in JavaScript can be invoked with any number of arguments, regardless of how many parameters the function is defined to accept; missing arguments are set to undefined and additional arguments are ignored.[2] To model this in the implementation, we define a common set constructor $\mathrm{Fun}_n()$ that, in addition to cons, this, return, and taint parameters, takes n arguments, where n is the maximum number of arguments across all function definitions and applications in the program. When a definition or call in the program uses fewer than n arguments, we pad the remaining arguments with fresh constraint variables. We omit these details from the presentation since they are straightforward.

Iterative Field-Sensitivity. Due to the flexibility of JavaScript objects, we must assume that every object can have a binding for every possible static field. However, this naïve approach does not scale, as the product of object count and field count is often very large. Instead, we perform an iterative field-sensitive analysis that tracks fields on a per-object basis as needed. For each object, we begin by tracking only the fields that we are certain will exist based on the object definition. After solving the set constraints under these assumptions, we check whether any objects flow into accesses of fields that were not being tracked. We add constructed terms for missing fields as appropriate, and incrementally solve for the constraints again.

Current Limitations. Tracking reads and writes of dynamic fields, *i.e.,* array or dictionary lookups, is significantly harder than tracking statically known field names. Modeling these accesses with a $\mathrm{Fld}_\top$ set constructor, where $\top$ represents unknown fields, makes the analysis unscalable, due to large numbers of accesses through integer fields (for array objects) and complex string expressions that compute precise names of HTML elements on the page. For the purposes of our analysis, we make the dynamically checkable assumption that dynamically created field names, *i.e.,* dynamically created array or hash table indices, do not clash with statically known field names.

Our current implementation also does not support several other features of JavaScript, but these can be directly captured within our constraint-based SIF framework. These include the with statement, which allows its body to be evaluated with a given object's fields temporarily brought into scope, and call and apply forms that allow the programmer to explicitly set the this parameter of a function call, which can be used to implement closure-based inheritance.

4.4 Residual Policy Generation

We now describe how we use our constraint-based flow analysis to compute residual policies for holes. Recall that the residual policy comprises a set of must-not read MNR and must-not write MNW variables and fields. Thus, at a high level, for confidentiality (resp. integrity) policies, our goal is to find variables to which (resp. from which) the sensitive information may flow, and then prohibit the client from reading (resp. writing) those variables.

However, it turns out that several subtleties arise due to the combination of higher-order functions, aliasing and the requirement that residual policy checking be efficient. We illustrate these issues using examples that motivate our algorithm for generating residual policies. For the following examples, suppose we wish to enforce the confidentiality policy stating that the document's cookie should not flow into a hole.

Functions. Intuitively, the residual policy needs to prevent the hole from reading any variable in the context that is tainted by the cookie. However, the residual policy must also prevent the hole from *writing* certain variables. To illustrate, consider the following context:

```
f = function(x) { ... }
f(document.cookie);
```

That is, the context contains a function f that is called with the cookie as a parameter. Next, suppose that a hole is filled with:

```
f = function(x) { post(x); }
```

That is, the hole redefines the function f with another function that broadcasts the argument x. If this code is dynamically loaded, it can *overwrite* the original "trusted" function f, and a policy violation will occur if the new function is called from the context. If we could re-analyze the entire source of the context and the hole at the client, then we would deduce that the call in the context flows the cookie into the formal x of the new function defined in the hole. However, performing client-side flow analysis on the entire code each time a hole is loaded would make residual policy checking prohibitively inefficient. Instead, we observe that when a function's arguments receive confidential values, we can guarantee confidentiality by ensuring the function itself is not *overwritten* by the hole. Thus, even for confidentiality policies, certain variables, namely those holding function values whose arguments have been tainted, should not be written by the hole. Dually, for integrity policies, the residual policy must also include both must-not-read and must-not-write sets.

Aliasing. Consider a context that contains the following code snippet, where tmp is not aliased to document:

```
z = tmp.cookie;
```

Hence, in the context, the value that flows into z is not sensitive. Next, suppose that a hole is filled with

```
tmp = document;
...
post(z);
```

That is, the hole aliases tmp and document and as a result, the assignment in the context can flow the confidential cookie into the variable z, thereby leaking the confidential information. Again, although re-analyzing the entire source on the client would detect this leak, it would be prohibitively expensive. One option is to treat the object x as confidential if x.f is confidential. Thus, we could treat document as confidential since document.cookie is confidential and prohibit any hole from reading document. However, this is far too restrictive as it is perfectly safe and common for the hole to read document and its non-tainted fields. Instead, when generating a residual policy, we conservatively assume that once field f of *some* object contains confidential information, then *all* fields named f contain confidential information, no matter what object they belong to. For our example, since document.cookie is confidential, we assume that tmp.cookie is confidential, and hence z is confidential, and so in the residual policy, we prevent the hole from reading z. Similarly, in this example, we would also prevent the hole from directly reading any field called cookie. Thus, we can make the residual policy checking robust and efficient, even in the presence of aliasing, by *unifying* the taint information of each field f across all the objects that contain f, and preventing the hole from accessing f.

We now describe our constraint-based algorithm that analyzes a context in order to compute the MNR and MNW sets corresponding to the residual policy. For clarity of exposition, we omit the indirect, real, and prototype wrappers, and we only describe how residual policies are generated for confidentiality policies – it is straightforward to extend the method to integrity policies.

Taint Propagation. To compute the MNR and MNW sets, we use two new covariant unary constructors $\mathrm{NR}()$ and $\mathrm{NW}()$, which correspond to *not-read* and *not-write* taints. We seed the analysis

[2] Whenever a function is called in JavaScript, the list of actuals provided is bound to a special variable called arguments available within the body. We model this behavior in our implementation.

by adding constraints that flow these new taint constructors into the variables of the confidentiality policy. In particular:

$$\text{for each } (\mathtt{x}, \bullet) \text{ in the policy,} \qquad \mathrm{NR}(\mathtt{c_x}) \subseteq \mathcal{X}_{\mathtt{x}}$$

where $\mathtt{c_x}$ is a special constant associated with $\mathtt{x}$. We use unary constructors with these special constants as arguments so that we can we can later define filter (*i.e.,* $\subseteq_{A,B}$) constraints. In addition to the basic flow constraints from Figure 8, which propagate these taint seeds throughout the context code, we add new constraints to account for the subtleties described above. In particular, to handle higher-order functions, we *contravariantly* (resp. *covariantly*) propagate the taints from the function arguments (resp. return values) to function definitions:

$$\begin{array}{ll} \text{for each fun-def labeled } i, & \mathcal{X}_{\mathtt{p}_i} \subseteq_{\mathrm{NR,NW}} \mathcal{X}_i \\ & \mathcal{X}_{\mathtt{p}_i} \subseteq_{\mathrm{NW,NR}} \mathcal{X}_i \\ & \mathcal{X}_{\mathtt{ret}_i} \subseteq_{\mathrm{NR,NR}} \mathcal{X}_i \\ & \mathcal{X}_{\mathtt{ret}_i} \subseteq_{\mathrm{NW,NW}} \mathcal{X}_i \end{array}$$

where $\mathcal{X}_i$, $\mathcal{X}_{\mathtt{p}_i}$, and $\mathcal{X}_{\mathtt{ret}_i}$ are the constraint variables representing the flows into the function labeled i, its formal parameter, and its return value respectively. Finally, to handle aliasing, we unify the taints across all objects containing a field $\mathtt{f}$ by creating a special variable $\mathcal{X}_{\mathtt{f}}$ and generating the following constraints:

$$\begin{array}{ll} \text{for each object labeled } i, & \mathcal{X}_{i.\mathtt{f}} \subseteq_{\mathrm{NR,NR}} \mathcal{X}_{\mathtt{f}} \\ & \mathcal{X}_{\mathtt{f}} \subseteq_{\mathrm{NR,NR}} \mathcal{X}_{i.\mathtt{f}} \\ & \mathcal{X}_{i.\mathtt{f}} \subseteq_{\mathrm{NW,NW}} \mathcal{X}_{\mathtt{f}} \\ & \mathcal{X}_{\mathtt{f}} \subseteq_{\mathrm{NW,NW}} \mathcal{X}_{i.\mathtt{f}} \end{array}$$

Residual Policy Generation. To compute the residual policy, we solve the entire set of constraints, that is, the basic flow constraints augmented with the constraints above. Intuitively, if a $\mathrm{NR}()$ (resp. $\mathrm{NW}()$) constructor flows into the constraint variable $\mathcal{X}_{\mathtt{x}}$ corresponding to the program variable $\mathtt{x}$, then the variable is added to the MNR (resp. MNW) set of the residual policy. Let S be the constraint solution. We write $S \vdash t \rightsquigarrow \mathcal{X}$ if the solution S maps the constraint variable $\mathcal{X}$ to a set containing the term t. The must-not-read and must-not-write sets of the residual policy are defined:

$$\begin{array}{rlr} MNR \doteq & \{ \mathtt{x} \mid S \vdash \mathrm{NR}(\cdot) \rightsquigarrow \mathcal{X}_{\mathtt{x}} \} \cup & \text{(Not-Read-Variables)} \\ & \{ \mathtt{f} \mid S \vdash \mathrm{NR}(\cdot) \rightsquigarrow \mathcal{X}_{\mathtt{f}} \} & \text{(Not-Read-Fields)} \\ MNW \doteq & \{ \mathtt{x} \mid S \vdash \mathrm{NW}(\cdot) \rightsquigarrow \mathcal{X}_{\mathtt{x}} \} \cup & \text{(Not-Write-Variables)} \\ & \{ \mathtt{f} \mid S \vdash \mathrm{NW}(\cdot) \rightsquigarrow \mathcal{X}_{\mathtt{f}} \} & \text{(Not-Write-Fields)} \end{array}$$

That is, a variable or field must not be read (resp. written) if the not-read taint (resp. not-write taint) constructor flows to the constraint variable corresponding to the variable or field.

Residual Policy Checking. To verify that a hole satisfies a residual policy, we perform a syntactic check that none of the variables or fields in MNR (resp. MNW) are read (resp. written) in the hole.

5. Evaluation

In this section, we describe experiments (Section 5.1) that validate three hypotheses about our approach: our information flow analysis using set constraints scales to real world JavaScript (Section 5.2); our staged information flow approach creates residual checks that are much smaller and faster than the full analysis, making them practical for running on the client side (Section 5.3); and our information flow analysis is precise enough to track useful properties (Section 5.4).

5.1 Experiments

We have implemented a static, constraint-based instantiation of the SIF framework for JavaScript. Our analysis is currently a standalone tool, not yet integrated within a browser. As a result, we do not have automatic support for staging when a script is loaded dynamically. Instead, for the purposes of evaluation, we have implemented a Firefox browser extension that intercepts all dynamic code loading calls, and *inlines* the new code in the surrounding context. The subsequent static analysis proceeds as if the dynamic content had been there originally. Once our analysis engine is combined with the browser, a dynamically loaded script will instead trigger the staged analysis.

We use the JSure parser [3] as a front-end to parse JavaScript source into OCaml abstract syntax trees, over which we generate constraints. We use the Banshee [20] constraint solver to build and solve constraints. The Firefox extension is written in approximately 500 lines of JavaScript, the Banshee bindings in 400 lines of C and OCaml, and the staged information flow tool in about 6,000 lines of OCaml.

Benchmarks, policies and holes. We used our Firefox extension to collect the closed-program source for all the web sites from the Alexa top 100 list [1]. Alexa is a company that tracks web page traffic, and generates the lists of the most popular 100 web sites by country and by language. We ran our staging engine on all 100 sites in the top 100 English pages.

We checked two policies on each web site: (1) a confidentiality policy stating that the cookie value should not flow into the hole, and (2) an integrity policy stating that no values from the hole should flow into the location bar. These policies are general enough that they apply to any web site, making it easier to systematically run on all the Alexa web sites.

For each web site, we systematically identified holes as any scripts originating from a different domain than the site's. Each closed-program we collected is a snapshot of whatever JavaScript executed on that particular run; subsequent visits to the same page would likely contain different dynamic code to populate the hole.

For each benchmark, we first ran our information flow analysis on the entire program. We then generated the residual policy for the holes that we identified, and performed the residual checks on the code in the hole. This simulated the situation where holes are not available at the first stage, but are made available at the second stage.

Summary of results. Of the 100 sites in the Alexa list, 97 had JavaScript, 64 had holes in them, and of the ones with holes in them, we were able to parse 63. Our full unstaged analysis successfully completed on all 63, and our staged analysis successfully completed on 62 of these. The one benchmark that our staged analysis failed on (by running out of memory while generating the residual policy) is the largest benchmark in the Alexa top 100, namely `wsj` with 43,698 lines of JavaScript (which is twice as large as the next largest benchmark).

5.2 Performance of Unstaged Analysis

Figure 9 plots lines of code vs. running time of the unstaged full analysis for the cookie confidentiality policy; the plot for the integrity policy follows similar trends. Our data shows that, for benchmarks up to 13,000 lines of code (which accounts for about 80% of the benchmarks) the running time does not grow very fast, and stays under twelve seconds. Nevertheless, these times are too slow to run on the client side each time that a new hole is filled. Beyond 13,000 lines of code, even though the running time grows much faster, our unstaged full analysis still scales to the largest of JavaScript programs in the Alexa top 100 (76.0 seconds for 43,698 lines of JavaScript). Most of the benchmarks that take over a minute

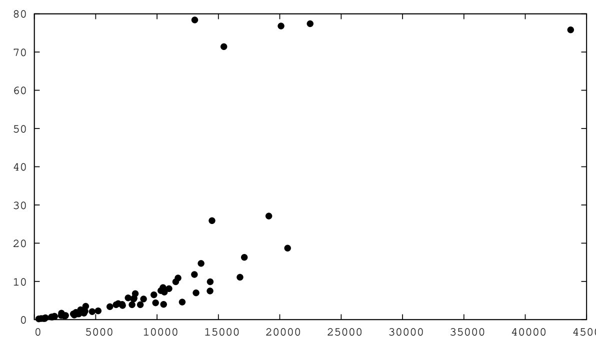

Figure 9. Analysis time (seconds) of unstaged analysis for cookie policy vs. lines of code.

to analyze make heavy use of prototypes. This observation points to an area of possible performance tuning for future work.

5.3 Performance and Benefits of Staging

Figure 10 shows some of the results from the 62 benchmarks on which our staged analysis ran. The last line of the table, however, averages over *all* benchmarks. The columns in the table are as follows: "Site and rank" gives the name of the web site and its rank in the top 100 list; "Total LOC" gives the number of lines of code on the web site, including the hole, as formatted by our own JavaScript pretty-printer; "Hole LOC" gives the number of lines of code in the hole; "Full" gives the time it took to run our unstaged information flow analysis on the entire JavaScript program; "GenRes" gives the time it took our analysis to generate the residual checks for the holes that we identified; "ChkRes" gives the time it took to perform the residual checks on the code from the holes; "FF" states whether or not the unstaged information flow analysis found any flow for the given policy (✓ indicates flow, × indicates no-flow); "RF" states whether or not the residual checks found any flow for the given policy. All times are in seconds.

In general, the time for computing residual policies is on the same order of magnitude as the time for running the unstaged full analysis. Even though the residual policy generation uses more kinds of constraints, it does not always take longer to solve, because only the context is being analyzed, which is smaller than the entire code that the unstaged analysis ran on.

Our data shows that all residual checks run under one second, and most run under one tenth of a second. The residual checks are performed in a single pass along with parsing, so in fact most of the time for performing residual checks lies in the parsing, which must be done anyway. Our current implementation uses a parser generator that is not optimized for speed, which leaves room for performance improvements.

As a result, residual checks would add only minimal overhead to run in a client browser, especially if we further tune the performance of our checker. Our data also shows that residual checks are about two orders of magnitude faster than the full analysis, which on average runs in 9.9 seconds, and thus would be too slow to run in a client browser. These observations together point to the benefit of staging: the full analysis would be too slow to run on the client browser, but if the developer could run the first stage of the analysis, then the remaining checks are fast enough to be run in the client browser.

5.4 Precision

In order to assess the precision of our unstaged analysis, we randomly sampled 17 of the benchmarks for the cookie-flow policy, of which 5 reported that the cookie does not flow into a hole. A manual inspection of these examples reveals that this is indeed the case. By looking at the code of the remaining 12 benchmarks, we determined that 8 of them contained holes that read, and even modified, the cookie. Many of these sites included scripts from popular ad services, such as GoogleSyndication and QuantServe, and data tracking services, like GoogleAnalytics. These services make use of cookies as a persistent storage for statistics across multiple page visits.

The reported flow on the remaining 4 benchmarks were false positives in our unstaged analysis, which were all caused by the lack of context-sensitivity. For example, if the cookie and an unrelated string both flow into the same function, and this function flows its argument to its return value, then both strings will flow to the returning call sites, smearing the actual flows. Techniques for extending set constraint-based program analysis with context sensitivity would help in this situation.

We evaluate the precision of our staged analysis by comparing its results ("RF" column in Figure 10) with the unstaged version ("FF" column). In general, the answer to whether the policy is violated should be the same in both unstaged and staged modes, and this is indeed the case for most of our benchmarks.

However, there are several benchmarks on which this is not the case. For confidentiality policies, there are 4 benchmarks for which the unstaged analysis finds no flow, but the staged one reports flow. For integrity, there are 8 such benchmarks. In each of these cases, the residual analysis reports a spurious flow because of how we conservatively taint fields when generating residual policies.

As expected, there are no cases in which the unstaged analysis reports flow but the staged analysis reports no flow.

6. Related Work

Static Information Flow. There is a rich literature on modeling security properties using information flow [15]. Many of these ideas are manifested as *static* language-based techniques for ensuring that the values of *high* security values do not flow into *low* security outputs. These include type systems [33, 25], Hoare-logics [5], and safety (model) checking [30]. Dually, there are techniques for checking that low-security (*i.e., tainted*) values do not flow to safety critical operations. These include the use of type qualifiers [28] and dataflow analysis [21]. Unfortunately, these techniques work on closed programs (or require summaries or stubs for missing code), and further, often rely on underlying structure like types, and hence cannot be applied directly to JavaScript.

Dynamic Information Flow. Several dynamic techniques for information flow control have been proposed at the language, operating system and architecture levels. The type system of [23] allows the specification and dynamic enforcement of richer flow and access control policies including the dynamic creation of principals and declassification of high-security information. These ideas cannot yet be applied in our setting as they require a closed system, written in a statically typed language (Java), and further, annotations must be provided to specify and verify the policies. There are several projects that use dynamic tainting, either via binary rewriting [24], at the architecture level [29, 32], or using virtual machines [8]. We leave the implementation of a dynamic instantiation of our framework, possibly for enforcing the residual policy checks, as an avenue for future work. However, we conjecture that dynamically tracking flows is likely to incur a significant run time overhead, and hence, is not a likely candidate for client-side deployment. Several recent projects [9, 37] propose expressive OS mechanisms for information flow control. Here the goal is to provide abstractions that allow application developers to specify policies about where

Site and rank	Total LOC	Hole LOC	Flow from cookie to hole?					Flow from hole to location bar?				
			Full	**GenRes**	**ChkRes**	**FF**	**RF**	**Full**	**GenRes**	**ChkRes**	**FF**	**RF**
3. `myspace`	22,469	3,484	77.4	27.4	0.52	✓	✓	105.3	37.2	0.52	×	×
4. `youtube`	7,187	779	3.7	4.4	0.20	×	×	3.6	4.8	0.18	×	✓
10. `aol`	4,714	255	2.1	2.9	0.06	✓	✓	2.1	3.4	0.06	×	×
13. `go`	904	60	0.5	0.9	0.03	×	×	0.5	0.9	0.03	×	×
15. `cnn`	15,445	3,472	71.4	18.0	0.52	✓	✓	83.1	30.4	0.54	×	✓
16. `espn.go`	7,155	28	4.0	7.0	0.03	×	×	4.0	8.2	0.03	×	×
18. `flickr`	747	713	0.3	0.1	0.12	×	×	0.3	0.2	0.12	×	×
24. `imdb`	556	13	0.3	0.5	0.02	×	×	0.3	0.6	0.02	×	×
28. `weather`	20,104	232	76.8	106.5	0.12	✓	✓	72.5	200.6	0.09	✓	✓
35. `foxnews`	13,589	70	14.7	30.7	0.10	✓	✓	15.0	50.6	0.04	×	×
42. `doubleclick`	3,259	1,203	1.5	1.2	0.21	✓	✓	1.4	1.4	0.21	×	×
43. `bbc.co.uk`	8,639	41	3.9	7.5	0.03	✓	✓	3.9	8.6	0.02	×	×
44. `walmart`	13,174	101	7.0	22.0	0.09	✓	✓	7.2	55.5	0.07	✓	✓
46. `rr`	2,545	70	1.1	1.8	0.05	✓	✓	1.1	3.3	0.03	×	×
47. `target`	10,532	61	4.0	7.0	0.04	✓	✓	4.0	8.1	0.04	×	×
48. `netflix`	9,879	27	4.4	8.4	0.03	×	×	4.5	9.8	0.02	×	×
49. `nfl`	10,485	170	8.4	16.8	0.03	×	×	8.4	21.2	0.03	×	×
57. `hulu`	14,476	545	25.9	42.1	0.11	✓	✓	28.2	131.4	0.12	×	×
58. `verizon.net`	3,456	167	1.5	2.1	0.05	✓	✓	1.5	2.4	0.04	×	×
62. `disney.go`	3,383	6	1.9	3.3	0.03	×	×	1.9	3.8	0.02	×	×
63. `bestbuy`	10,975	3,916	8.2	10.6	0.76	✓	✓	8.7	290.0	0.80	×	✓
64. `msn.foxsports`	6,838	490	4.2	7.0	0.14	✓	✓	4.2	16.1	0.18	×	✓
67. `cnet`	10,598	242	7.2	22.6	0.17	✓	✓	7.3	29.2	0.06	×	×
71. `linkedin`	7,964	1,816	3.9	3.4	0.32	×	×	3.9	3.6	0.29	✓	✓
75. `gamespot`	13,041	1,491	11.8	23.6	0.32	✓	✓	11.5	30.6	0.28	×	×
77. `veoh`	9,742	86	6.5	13.1	0.07	×	✓	6.6	32.6	0.04	×	×
79. `latimes`	8,225	55	6.8	9.9	0.04	✓	✓	6.8	11.7	0.03	×	✓
80. `nbc`	7,644	74	5.8	8.5	0.04	×	×	5.8	10.4	0.04	×	×
87. `reuters`	4,049	258	1.7	2.4	0.06	✓	✓	1.7	2.6	0.06	×	×
88. `imeem`	12,050	194	4.6	7.9	0.04	×	×	4.6	8.7	0.04	×	×
89. `gamefaqs`	365	77	0.2	0.3	0.03	×	×	0.2	0.3	0.03	×	×
90. `tinypic`	6,658	64	3.9	6.0	0.03	×	×	4.0	6.6	0.03	×	×
92. `abcnews.go`	14,330	246	9.9	18.0	0.07	×	✓	9.8	21.4	0.08	×	✓
99. `dailymotion`	11,709	379	10.9	19.3	0.08	×	✓	10.8	30.4	0.08	×	×
100. `people`	6,152	261	3.4	4.8	0.07	✓	✓	3.4	6.8	0.06	×	×
Average	**7,979**	**597**	**9.9**	**14.0**	**0.13**			**10.7**	**28.4**	**0.12**		

Figure 10. Sample results from Alexa web sites with holes. Average numbers are for all benchmarks (including those not in the table), and times reported are in seconds.

data generated by the process should be allowed to flow. These approaches are too coarse-grained to be applicable to our setting.

Analyzing JavaScript. Several authors have studied the problem of analyzing JavaScript. Some of the idiosyncratic features of JavaScript are described in [31], which also presents a type system for statically *checking* JavaScript programs. Further, [6] describes an algorithm for *inferring* types for JavaScript programs. However, it is unclear whether JavaScript programs in the wild satisfy the typing disciplines described in these works. Neither approach deals with dynamically generated code, and hence cannot directly be applied to our setting. The interaction of JavaScript and web browsers is studied in [36], which presents a formal semantics of the interaction, and uses it to describe a general framework for dynamically verifying arbitrary safety properties inside the browser. Gatekeeper [22] is a static analysis framework for JavaScript that focuses on performing analysis in a single stage (*e.g.,* on the server). In contrast, our primary focus is on developing residual checks that specify how dynamically loaded code should behave in order for the system to satisfy high-level flow policies.

Web and Browser Security. Several recent projects have considered the problem of securing web applications via browser and language mechanisms. Many vulnerabilities arise from not appropriately *sanitizing* user generated content on the server side. Several server-side tools apply static analysis to determine whether user generated content has been properly vetted [19, 35, 34]. To ensure safety on the client side, one simple and elegant approach is to only allow previously known and authorized scripts to run on a web page [18]. Unfortunately, this makes it harder to use dynamically generated third-party content, and hence is not applicable in our setting. Finally, there have been several proposals for redesigning the ecosystem within which web-applications are built and deployed [4, 2, 7]. In essence these approaches advocate that web-applications be built in higher-level languages like $C^\sharp$, Java and JIF respectively, thereby availing of the protection mechanisms available in those languages. It remains to be seen whether web-application developers are willing to trade the flexibility and rapid-prototyping strengths of JavaScript for the security benefits offered by strongly typed languages.

Set Constraint-based Program Analysis. Set constraints provide an expressive framework within which many kinds of program analyses including points-to analyses [14, 16], type qualifier inference [13], race detection [26], and uncaught exceptions [10]. Our contribution is to show that this expressive framework is especially suited to capturing the complexities of JavaScript including fields and higher-order functions, and that after using optimizations like the optimistic field analysis the resulting analysis scales to the JavaScript that powers most popular websites.

References

[1] English: Alexa top 100 sites, November 2008. `http://www.alexa.com/`.

[2] Google web toolkit, November 2008. `http://code.google.com/webtoolkit/`.

[3] Jsure, November 2008. `http://www.jsure.org/`.

[4] Volta, November 2008. `http://live.labs.com/volta`.

[5] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *SAS*, pages 100–115, 2004.

[6] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP*, pages 428–452, 2005.

[7] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP*, pages 31–44, 2007.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[9] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP*. ACM, 2005.

[10] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *SAS*, pages 114–126, 1997.

[11] M. Fähndrich, J. S. Foster, A. Aiken, and J. Cu. Tracking down exceptions in standard ml programs. Technical report, EECS Department, UC Berkeley, 1998.

[12] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

[13] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*. ACM, 1999.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS*, 2000.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[16] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, 2007.

[17] D. Herman and C. Flanagan. Status report: specifying javascript with ml. In *ML*, pages 47–52, 2007.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[19] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[20] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, pages 218–234, 2005.

[21] M. S. Lam, M. Martin, V. B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *PEPM*, pages 3–12, 2008.

[22] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. Technical Report MSR-TR-2009-16, Microsoft Research, Feb. 2009.

[23] A. C. Myers. Programming with explicit security policies. In *ESOP*, pages 1–4, 2005.

[24] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[25] F. Pottier and V. Simonet. Information flow inference for ml. In *POPL*, pages 319–330, 2002.

[26] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*. ACM, 2006.

[27] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots*, 2007.

[28] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[30] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[31] P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.

[32] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[33] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *POPL*, 2000.

[34] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[35] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.

[36] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL*, pages 237–249, 2007.

[37] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.