First-Class Labels: Using Information Flow to Debug Security Holes*

Eric Hennigan Christoph Kerschbaumer
Stefan Brunthaler Per Larsen Michael Franz
{eric.hennigan, ckerschb, s.brunthaler, perl, franz}@uci.edu

University of California, Irvine

Abstract. We present a system of first-class labels that assists web authors in assessing and diagnosing vulnerabilities in web applications, focusing their attention on flows of information specific to their application. Using first-class labels, web developers can directly manipulate labels and express security policies within JavaScript itself, leveraging their existing knowledge to improve the quality of their applications. Introducing first-class labels incurs no additional overhead over the implementation of information flow in a JavaScript Virtual Machine, making it suitable for use in a security testing environment even for applications that execute large amounts of JavaScript code.

1 Motivation

The JavaScript programming language has become indispensable for Web 2.0 applications and powers almost all of today's banking and electronic commerce sites. These organizations regularly use JavaScript to process sensitive information, such as credit card numbers and user credentials. The ability to perform client-side processing has facilitated the adoption of interactive pages, while simultaneously introducing a new code injection attack vector known as Cross Site Scripting (XSS). Within the web browser, the JavaScript execution model allows objects from different domains to reference each other. This architectural weakness gives adversaries the ability to gain access to sensitive data held within the browser and manipulated by a page's code.

Currently, web sites rely on the browser enforced Same Origin Policy [14], which limits interactions between different domains, with the intent of separating content from different providers. This restriction applies to separate pages

* This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contract No. D11PC20024, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate, Sierra Vista Branch, the National Science Foundation, or any other agency of the U.S. Government.

and `iframe`'s, but does not prevent method and memory access when the host page includes a third party script, such as the JQuery library or a syndicated advertisement. The lack of isolation between scripts from separate origins that execute on the same web page, threatens the privacy of all web users.

Other systems implementing information flow within a web browser [9,10,13] attempt to perform fully automatic labeling, with no feedback from the developer. Because privacy concerns are application specific, we present an alternative approach that additionally provides developers with the ability to selectively focus on specific information flows within their web application. Without domain knowledge, fully automatic frameworks detect and report information flows, such as requests from content distribution servers, that application developers would prefer to disregard.

Based on our experience, we think that information flow tracking shows more promise as a web application security debugging tool, if it can help the developer focus only on flows relevant to an XSS vulnerability. We achieve this goal by extending an existing information flow tracking browser with a system of first-class labels that developers can use to inspect their application. By selectively tagging only those variables considered security sensitive, developers can focus their attention on flows of specific information, and avoid sifting through the morass of reports generated by automated tracking systems. We envision web developers using the first-class labeling system as part of a testing environment to answer common auditing questions: "Does this sensitive data ever influence a network request?" and "What values does this object influence?"

After presenting the threat posed by attackers (Section 2), we establish information flow terminology (Section 3) to clarify the capabilities of the underlying tracking engine on which we base our work. We then introduce details of the supporting information flow framework (Section 4) relevant to the following contributions:

- We extend JavaScript's syntax and semantics (Section 5) introducing a reflective `FlowLabelObject` and new `labelof` operator.
- To the best of our knowledge, we are the first to provide a first-class labeling system within JavaScript (Section 6) that allows developers to selectively tag application specific sensitive information from a webpage and compose security policies in JavaScript.
- We demonstrate the utility of the first-class labeling system by showing an attack that aims to exfiltrate sensitive user information and a JavaScript-specified network policy that stops the attack (Section 6.3).

We evaluate (Section 7) our first-class labeling system demonstrating that it maintains performance, resists JavaScript-level attacks against itself while exposing underlying security data structures, and provides a mechanism that the web developer can effectively use to debug security holes in a web application.

2 The Attacker's Threat

Throughout this work, we assume that the attacker has already injected code into the developer's web application. The attacker exploits an XSS vulnerability to inject code in the developer's web application, supplying a JavaScript payload via an included advertisement, mashup content, or library, or via an unsanitized form or URL. Although we limit the attack payload to JavaScript, we assume that its origin does not make it distinguishable from the rest of the web application's JavaScript codebase. The attacker also has publicly-facing knowledge about the application, obtained by visiting and interacting with the application and observing its behavior, which can be used to craft the payload. We also assume that the attacker controls their own web server.

These abilities combine to pose an information leak threat. The code injected into the web application executes with the full abilities of that application. The attacker crafts the payload to exfiltrate application sensitive information, such as personal login credentials, text the user enters into forms, or anything the web application displays to a visitor. The pilfered information leaves the application as part of a resource request submitted to the attacker controlled server, circumventing the Same Origin Policy.

For a typical example, exfiltration code embeds the sensitive data into a URL and attaches that URL to the `src` attribute of a payload generated `img` element. The web browser automatically issues a GET request for the image targeting the attacker controlled server. The attacker then reviews server request logs to harvest the exfiltrated information.

2.1 The Developer's Response

Knowing that the origin of attacker code does *not* reliably distinguish it from the rest of the web application, we focus on the malicious *behavior* of any code within the application. Indeed, an information leak might be the unintended result of a careless or uninformed application developer, rather than an attacker.

In response to this threat, a security-conscious developer tests their application in a web browser that monitors the flows of information within the application. To assist the developer in focusing their debugging attention on specific pieces of sensitive data within the application, we present a labeling system as a first-class language construct. Without leaving JavaScript, the developer creates a label and applies it to the sensitive data, tagging it with a unique identifier. The underlying information flow engine tracks the interaction of application (and injected) code with this sensitive data, ensuring that exfiltration code does not drop the label.

We present a mechanism that allows the developer to write a network monitor using JavaScript, so that they may observe a leak of information tagged as sensitive. The developer implements their own network monitor logic to inspect the labels of all resource requests, enabling the detection and debugging of an information leak.

Category	Descriptor	Example	Flow	Required Analysis
Explicit	Direct	`b = a`	a ⇒ b	Dataflow
	Indirect	`b = foo(_, a, _)` `c = bar(_, b, _)`	a ⇒ c	Dataflow (transitive)

Implicit	Direct	`if (a)`
		`  b = 1`
		`else`
		`  b = 0`
			a ⇒ b	Control Flow (dynamic)

	Indirect	`c = true`
		`if (a)`
		`  b = false`
		`if (b)`
		`  c = false`
			a ⇒ c	Control Flow (static)

Table 1. Terminology of Information Flows.

3 Information Flow Terminology

Previous research in the field of information flow applied to dynamic languages reveals a need for clarifying terminology that goes beyond the basic categories introduced by Denning and Denning [5]. We follow this trend by extending the established categories with easy-to-remember descriptors. We intend for the terminology introduced here to bring clarity and precision to the research describing information flow systems, especially research targeting dynamic languages. The more refined terminology allows us to characterize the capabilities of the information flow tracking engine (Section 4) which supports the first class label system introduced in this paper (Section 6).

3.1 Explicit Information Flows

An *explicit flow* occurs as a result of a dataflow dependence. Table 1 breaks this category down into two descriptors: *direct*, corresponding to an immediate dependence, and *indirect*, corresponding to a transitive dependence.

Explicit Direct Flows occur when a value is influenced as a result of direct data transfer, such as an assignment. A simple single-statement, intra-procedural dataflow analysis can identify these flows. Subexpressions involving more than one argument also have a direct explicit information flow from all argument values to the operator's resulting value. Any labeling or tagging framework that tracks security type information across direct explicit flows includes basic semantic rules for label propagation in each of the language's operators.

Explicit Indirect Flows occur as the transitive closure of direct flows. Identification of indirect flows requires more powerful multi-statement or inter-procedural dataflow analysis. The code example for indirect flows in Table 1 shows the transitive nature of this analysis via a functional dependence between values. This paper preserves the use of the term "indirect" as originally defined by Denning and Denning [5].

3.2 Implicit Information Flow

An *implicit flow* occurs as a result of a control-flow dependence. Table 1 breaks this category down into to descriptors: *direct*, corresponding to a runtime dependence, and *indirect*, corresponding to a static dependence.

Implicit Direct Flows occur when a value depends on a previously taken control-flow branch *at runtime*. Identification of this dependence requires a tracked program counter and a recorded history of control-flow branches taken during program execution. Presently, systems that track the program counter to propagate dependence information are known as "dynamic information flow tracking" systems.

Implicit Indirect Flows occur when a value depends on a control-flow branch *not taken* during program execution. Identification of this dependence requires a static analysis prior to program execution. Because the dependence follows code paths not taken at runtime, these flows are notoriously difficult to detect in dynamic programming languages. Unfortunately, even static languages include features, such as object polymorphism and reference returning functions, which make the receiver of an assignment or method call unknown at compile time. Dynamic programming languages, such as JavaScript, include first-class functions, runtime field lookup along prototype chains, and the ability to load additional code at runtime via `eval`. These features prohibit even a runtime analysis from identifying all the values possibly influenced in all alternative control-flow branches.

4 Supporting Framework

The framework which supports the first-class labeling system presented in this paper implements dynamic information flow as part of the JavaScript Virtual Machine (VM). Any viable information flow system within a web browser must support runtime creation and application of labels because security principals represented on a web page do not become known to the browser and JavaScript VM until a user visits the page. Every JavaScript value carries a label representing an element from the finite powerset lattice over principals. The VM conservatively labels the result of every operation with the union (join) of the labels of its inputs, monotonically moving up the lattice of security principals. To prevent attack code from removing or downgrading the labels applied to values tracked by the VM, the labeling framework does not currently provide a mechanism for declassification (i.e., it does not expose an intersection (meet) operation).

4.1 Storage of Security Principals and Labels

The underlying labeling framework allows any JavaScript value to be used as a security principal. Our first-class labeling system merely exposes this ability as a concise labeling API to the JavaScript developer. As we shall see (Section 6),

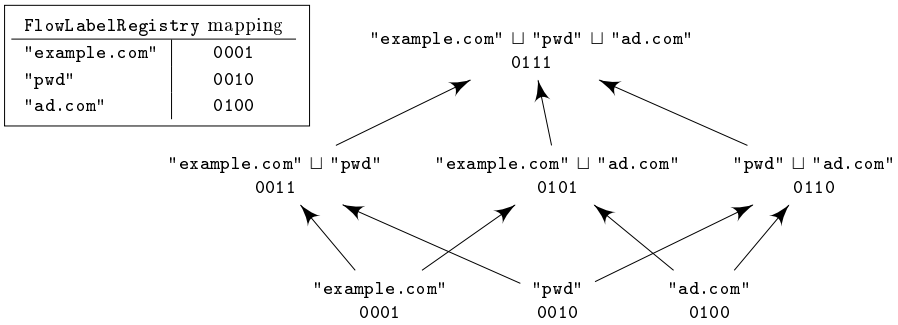

Fig. 1. The `FlowLabelRegistry` mapping three JavaScript strings used as security principals to unique bit positions. These principals form a lattice of security labels, represented as bit vectors.

the ability to use any JavaScript value as a principal gives web authors enough power to represent security principals as a native part of an application's code.

The supporting information flow VM interns every JavaScript value used as a security principal in the `FlowLabelRegistry`, mapping it to a unique bit position. Figure 1 depicts the interning of three JavaScript string objects, `"example.com"`, `"pwd"`, and `"ad.com"`, each representing a security principal in the `FlowLabelRegistry`. To minimize the attack surface on the system itself, our first-class extensions (Section 5) do not make this data structure accessible to the JavaScript programmer.

As shown in Figure 1, the mapping held by the `FlowLabelRegistry` allows a bit vector to represent each security label. The supporting VM attaches to every JavaScript value a security label, representing an element from a powerset lattice over security principals. Current implementation of the underlying information flow framework does not support more than 64 unique principals. However, we have not found this to be a problem in practice (Section 7).

4.2 Label Propagation

Our labeling system rests atop a pre-existing, JavaScript information flow VM that provides every JavaScript primitive and object reference with a security label. The supporting VM propagates labels through data flows and maintains a shadow stack of labels attached to the program counter [8] that tracks influence through control-flow transfers taken at runtime. These mechanisms allow it to track up to implicit direct information flow (as defined in Section 3).

Performing information flow tracking at the VM level allows the supporting framework to avoid potential attacks on the tracking system itself. This design reduces the attack surface compared to JavaScript rewriting systems [4, 9].

Exposing the underlying framework through our first-class labeling system might create a new attack surface (targeting the underlying label framework itself) meant to be hidden by design. As a result of this concern, we chose

not to support declassification through our first-class labeling system. Both the JavaScript developer and any potential JavaScript attack code can only create, apply, and inspect labels, but cannot remove them.

```
function sniffPassword(pw) {
  var spw = "";
  for (var i = 0; i < pw.length; i++) {
    switch(pw[i]) {
    case 'a': spw += 'a'; break;
    case 'b': spw += 'b'; break;
    ... // other characters elided
    }
  }
  return spw;
}
```

Listing 1.1. Password sniffing via implicit direct information flow.

Listing 1.1 gives an example of an attacker provided function which attempts to drop any label attached to the argument `pw`. The existing label framework can track the control-flow dependence of the return variable (`spw`) on the argument (`pw`) at both the loop condition (`pw.length`) and the switch condition (`pw[i]`). By performing such tracking, the returning variable `spw` subsumes the same set of principals as the incoming function argument `pw`. The tracking and propagation engine prevents the attacker from dropping labels through implicit direct information leaks in exfiltration code.

4.3 Information Flow in the Browser

Our first-class labeling system resides in a web browser that consists of a hosted JavaScript VM and additional subsystems for information storage, rendering, document description, and network communication. These other subsystems represent covert channels through which an attacker may communicate information. Currently, the supporting framework automatically applies labels to dynamically loaded code and resources according to the site of origin.

In addition to storing visible page elements, the Document Object Model (DOM) allows creation of invisible elements within the document that can be used to store and communicate information. The supporting framework propagates labels to HTML elements and attributes within the DOM so that an attacker cannot use it as a channel to remove labels.

The information flow tracking web browser also contains a network monitor that observes the labels on all network traffic: dynamic requests for remote resources such as images and stylesheets, HTTP GET and POST methods for forms, and `XmlHttpRequest` for AJAX. Our first-class labeling system presents to the web developer a mechanism for registering JavaScript functions which implements network monitor logic, enabling the developer to inspect labels attached to resource requests and thereby discover information leaks.

5 Design and Implementation of First-Class Labels

Before discussing the first-class label interface that a JavaScript developer uses to hook into the supporting information flow framework, we first give details explaining the extensions and modifications necessary to support labels as first-class JavaScript objects.

5.1 Reflecting Labels into JavaScript

The supporting framework contains a `FlowLabelRegistry` that maps primitive values and JavaScript objects used as principals to a position within a bit vector label. By holding a reference to every JavaScript object (within the standard heap) used as a principal, the `FlowLabelRegistry` keeps it alive during garbage collection. Because of the limited number of principals which can exist within the system (Section 4.1) the `FlowLabelRegistry` does not release any principals.

Our first-class labeling system reflects the underlying labels into the JavaScript language, as native JavaScript objects, via a `FlowLabelObject` wrapper. When reflected into JavaScript as `FlowLabelObject` instances, security labels can themselves be labeled and can also act as security principals, just like any other JavaScript value. Additionally, they are callable objects, providing an interface to apply the internally stored label onto any given argument value. In the interest of clarity, we do not use any examples that exhibit the inherent recursive nature of the first-class labeling system.

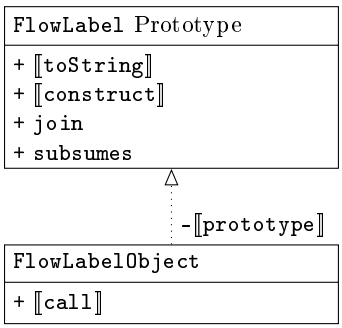

Fig. 2. UML class diagram of the first-class labeling system. Our system introduces the `FlowLabel` prototype constructor, and `FlowLabelObject` instances. As in the ECMA [6] language standard, [[•]] indicates implementation internal methods.

Our first-class labeling system also introduces a singleton `FlowLabel` prototype, which both holds methods common to all `FlowLabelObject` instances and provides an interface through which the JavaScript developer can construct `FlowLabelObject`s. Figure 2 uses UML to depict the relationship between the `FlowLabel` prototype singleton and `FlowLabelObject` instances.

5.2 JavaScript Syntax Extension to Retrieve Labels

Our first-class labeling system implements a small change to the JavaScript language permitting JavaScript code to retrieve a label from a given value. We introduce the keyword `labelof`, as a new case in the *UnaryExpression* grammar rule of the ECMA [6] language standard. Figure 3 presents the entire grammar rule, including our new language keyword.

UnaryExpression:
    PostfixExpression
    `delete` *UnaryExpression*
    `void` *UnaryExpression*
    `typeof` *UnaryExpression*
    `++` *UnaryExpression*
    `-` *UnaryExpression*
    `+` *UnaryExpression*
    `-` *UnaryExpression*
    `~` *UnaryExpression*
    `!` *UnaryExpression*
    `labelof` ***UnaryExpression***

Fig. 3. Modified JavaScript grammar rule for *UnaryExpression*. Our first-class labeling system introduces the `labelof` keyword.

5.3 Network Hook in the Web Browser

To permit the enforcement of policies written in JavaScript, we make one additional change to the web browser hosted JavaScript environment. Our first class labeling system exposes the underlying network monitor through a function, `registerSendMonitor(fn)` on the hosted `navigator` object. Using this feature, the web developer can phrase application specific security policies concerning allowed network communication as a JavaScript function within the web application itself. Once registered, these functions act as network monitors that inspect the payload of all resource requests before being sent over the network.

6 Using First-Class Labels

We design the first-class labeling system and its JavaScript API according to the functional programming paradigm, with the purpose of making it easier for web developers to adopt. The first-class labeling system contains one minor syntax change to the JavaScript grammar, introducing the new `labelof` operator and keyword. The system also extends the hosted environment (*not* the ECMA specification) with a new built-in `FlowLabel` prototype constructor object that holds methods for label composition (`join`) and comparison (`subsumes`). Labels take the form of native built-in `FlowLabelObject` instances, and behave with the

same semantics as any other JavaScript object. Our first-class labeling system makes a minimal set of changes necessary to expose the underlying information flow framework.

We show how our framework detects and prevents information leakage that might occur due to a script injection attack. In the following examples we show output of our system at the JavaScript console. All statements executed by the console begin with a '>'. The console describes the resulting value in two parts: the value itself and the label attached to that value.

6.1 Label Creation

Our system introduces a `FlowLabel` prototype singleton to the JavaScript environment hosted by the web browser. This object implements the internal [[`construct`]] method so that JavaScript code may create first-class label objects. The web developer may choose any valid JavaScript value to act as a security principal, and pass that value into the constructor. After interning the provided value in the underlying framework's `FlowLabelRegistry`, the constructor returns a `FlowLabelObject` instance. Interning the principals allows unique identification of labels held by `FlowLabelObject` instances. In the interest of avoiding attacks on the labeling system itself, our system does not provide programmatic access to the `FlowLabelRegistry`.

```
1 > pwdLabel = new FlowLabel("pwd");
2   [FlowLabelObject pwd] [FlowLabel example.com]
```

Listing 1.2. Creating a Label Object.

Listing 1.2 shows a web developer creating a label using the JavaScript string, `"pwd"`, as a security principal. The underlying information flow framework automatically applies a label to every resource representing its domain of origin. Consequently, the resulting `FlowLabelObject` instance returned from the constructor itself carries a label representing the origin of this code snippet: `example.com`.

6.2 Label Application

The `FlowLabelObject` instance acts as a first-class wrapper object around an internal bit-vector representation of a security label. The `FlowLabelObject` instance also implements the internal [[`call`]] method, so that the security label may be attached to other JavaScript values, When the `FlowLabelObject` functor is passed a value, it unions that value's current label with its internally stored label and returns the result.

```
3 > pass = "24sk09nk12";
4   24sk09nk12 [FlowLabel example.com]
5 > pass = pwdLabel(pass);
6   24sk09nk12 [FlowLabel pwd, example.com]
```

Listing 1.3. Applying a Label to a JavaScript Value.

Listing 1.3 shows the JavaScript developer applying the password label constructed previously (Listing 1.2), `pwdLabel`, to a string, `pass`. After label application, the resulting password string carries a label describing both the domain of origin, `example.com`, and the password security principal, `"pwd"`.

6.3 Label Retrieval and Comparison

We now assume that the attacker injects code using `sniffPassword` (Listing 1.1) in an attempt to drop the label of the user's password. Because the underlying framework tracks labels inter- and intra-procedurally with respect to both data and implicit direct control flows (Section 4.2), the label on the resulting sniffed password carries both the attacker's principal and the user's password principal. Our first-class labeling system exposes the network object to JavaScript, allowing interception of the information leak at the time of a network request.

```
1 navigator.registerSendMonitor(
2   function(method, url, payload) {
3     if (method == 'GET') {
4       var lab = new FlowLabel("example.com");
5       lab = lab.join(new FlowLabel("pwd"));
6
7       if (!lab.subsumes(labelof url))
8         log(url + " has unexpected label");
9       if (!lab.subsumes(labelof payload))
10        log(payload + " has unexpected label");
11    }
12    // other types of network request elided
13    return true;
14  });
```

Listing 1.4. Developer Provided Network Monitor Function.

Suspecting a possible information leak, the web developer implements a network monitoring logic in a JavaScript method, and registers it through the `navigator.registerSendMonitor` method. When the attack code attempts to communicate the pilfered information over the network, our labeling system first executes all registered monitors (in registration order) to determine if the request conforms to the developer-specified policy.

Listing 1.4 shows an example network monitor that takes advantage of the labels automatically applied by the underlying information flow framework. On Line 4, the developer creates a label representing the security principal, `example.com`. The `FlowLabelRegistry`'s interning of principals ensures that any labels created in this monitor function exactly match the same labels created elsewhere.

Through prototype-based inheritance, all `FlowLabelObject` instances have a `join` method that returns a new `FlowLabelObject` instance representing the union of its argument `FlowLabelObject` instances. On Line 5 of Listing 1.4, the developer `join`s the security principal `example.com` with `"pwd"` to compose together existing labels into a single label representing the union of all principals the developer wishes to allow in an HTTP GET request.

Information flow propagation within the VM labels each new value with the join of the labels of the arguments used to construct that value. Consequently, label propagation naturally results in values labeled with more than one principal, even when the original program only seeded a few values, each with a single principal. In response to this phenomenon, our developer uses the `subsumes` method (Line 7 and Line 9 of Listing 1.4) to check that the label of the request is a subset of all allowed principals. Although our first-class label wrappers also permit strict equality comparison (JavaScript operator ===) between two `FlowLabelObject` instances, we strongly encourage using the `subsumes` relation for expressing security policy constraints using subsets of principals. This practice allows catching all values with labels below the given upper bound (supremum).

Our labeling extension introduces the `labelof` operator so that JavaScript code can retrieve labels attached to variables for inspection and application. On Line 7 and Line 9 of Listing 1.4, the developer uses this operator to obtain the label attached to the target request `url`, and network `payload`. Because the underlying framework propagates labels following data flows, the resulting `FlowLabelObject` instance returned from `labelof` operator is itself labeled with the union of the provided argument and current program counter. If desired, the developer may use the resulting `FlowLabelObject` instance to label other values.

In the example shown in Listing 1.4, the developer constructs a label over the password principal, `"pwd"`, at two different code locations: once to label the user's input and again in the network monitor. This practice causes no problem for our system, because the `FlowLabelRegistry` interns principals, allowing our system to consider identical, two `FlowLabelObject` instances constructed in different code locations but with equivalent JavaScript values.

7 Evaluation

To evaluate the effectiveness of our system for security debugging we examine four dimensions:

Performance. We show that underlying information flow framework is fast compared to other work and argue that the first-class labeling system introduces negligible overhead.

Completeness. The labeling system inherits the code coverage of the supporting information flow framework.

Security. We argue that the labeling system revealed to the JavaScript programmer does not present a new attack surface in any significant way.

Usability. We demonstrate how developers can use the system to debug security vulnerabilities in their web applications.

We evaluate the effectiveness of our system as a web application security debugging tool. We measure the robustness and performance of the underlying labeling framework, demonstrating that even sites with large libraries of JavaScript code present no execution difficulties. We also use the first-class labeling system to find and debug an XSS vulnerability.

7.1 Performance

The supporting framework, termed FlowCore, modifies WebKit's JavaScript engine JavaScriptCore (version 1.4.2) to attach labels to every value. Additionally, it contains data structures relevant for mapping label bits within a label to domains (the `FlowLabelRegistry`) and for propagating implicit direct information flow dependencies. To evaluate the costs imposed by FlowCore, we test it against an unmodified JavaScriptCore of the same version.

Because FlowCore implements tracking only in the interpreter, we execute both JavaScript engines with just-in-time compilation disabled. A dual Quad Core Intel Xeon 2.80 GHz with 9.8 GiB RAM running Ubuntu 11.10 executes all benchmarks (at niceness level -20). We choose to use the SunSpider [17] benchmark suite because its status as the standard benchmark suite for JavaScript makes it suitable for comparisons to other work. SunSpider includes test cases that cover common web practices, such as encryption and text manipulation. This benchmark test provides a measure of the baseline overhead involved in maintaining information flow data structures and propagating labels.

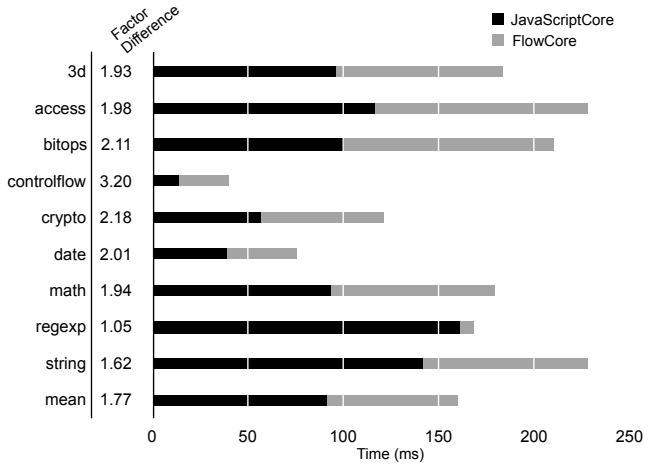

Fig. 4. SunSpider Benchmark results: JavaScriptCore vs. FlowCore.

Figure 4 reveals overall execution speed of JavaScript benchmark results: the mean execution time of FlowCore is 158.33 ms whereas the mean execution time of JavaScriptCore is 89.44 ms. The SunSpider benchmark does not contain first-class labeling operations, so the overall 77% slowdown represents the overhead incurred by the supporting framework's implementation of label propagation. In comparison, other information flow approaches [10] introduce a 150% slowdown making programs two to three times slower.

The VM stores labels as bit vectors attached to values and performs label propagation via bitwise-or. This representation ensures that first-class label ob-

jects are only present when explicitly constructed (`new FlowLabel`) or retrieved (`labelof`) by the developer. As a result, the introduction of the first-class labeling system into the hosted environment incurs no additional runtime performance overhead compared to a fully automatic labeling system. We do not evaluate the performance impact of the network hook, because it is insignificant within a debugging environment and the developer has the power to implement any monitor function they desire.

The performance of the underlying labeling framework implies that even sites with large amounts of JavaScript code execute without noticeable slowdown. To test whether the information flow tracking framework causes a noticeable performance decrease, we visited (and logged into) JavaScript intensive sites, such as Facebook, GMail, Google Maps, Bing, GitHub and Cloud9 IDE. These sites do not make use of the first-class labeling system introduced in this paper. However, user interaction proves that the performance overhead of the labeling framework does not introduce any usability issues.

7.2 Completeness

To verify that the underlying framework does not introduce any runtime bugs when interpreting either machine-generated or human-written JavaScript found in the wild, we automated the visiting of all sites in the Alex Top 500 [1]. This webcrawler injects code into each page, to perform two actions: (1) attach a network monitor and (2) fill out and submit the first form on the page using data labeled with an identifying principal. The injected monitor verifies that the submitted form generates a request containing the identifying principal.

Not only do we verify the label propagation engine against code in the wild, but we also use the first-class labeling system to develop a suite of unit test cases for ensuring the semantic correctness of the underlying labeling framework. Without first-class labels, we would be far less confident of the semantic correctness of the underlying framework's implementation of label propagation.

7.3 Security

The underlying framework, FlowCore, generates, at runtime, new security principals for every unique label generated by the developer and new domain encountered by the web browser. Introduction of runtime principals requires mutation of the `FlowLabelRegistry`. By design, FlowCore does not support declassification, preventing a communication channel via the labeling framework itself.

Our first-class labeling system exposes, to the web application and any injected code, a JavaScript API for creating and applying labels to JavaScript values. This exposure represents a new attack surface that might allow an attacker to target the labeling framework. However, we envision the web developer using the first-class labeling system only in a testing environment, where it provides no benefit to the attacker. Nevertheless, the lack of declassification means that the attacker-injected code cannot drop labels applied by the developer for debugging purposes.

Finally, our system allows registration of many monitor functions, through a JavaScript interface accessible by code injected into the web application. Our labeling system evaluates all monitor functions registered, in registration order. The developer-supplied monitor function always executes, even if the attacker's injected code happens to register a different monitor function first.

7.4 Utility as a Debugging Tool

To evaluate our first-class labeling system as a tool for testing web applications and discovering security vulnerabilities, we create a web page that contains a user login form. Acting as a malicious developer, we insert code into the page, which uses the `sniffPassword` label dropping code prior to exfiltrating the form contents to a second server via both an `XmlHttpRequest` and as part of an `img.src` URL. Acting as a security researcher, we mirror the page and add labeling code that applies a tag to the form's DOM node and a network monitor function that checks for the unique tag. Visiting the mirrored page successfully triggers the monitor function, alerting us to the exfiltration. WebKit's developer tools assisted us with finding the portion of the page responsible for generating the image request.

For a more realistic example, we attempt a similar attack using a mirrored `ebay.com` page obtained from XSSed [11], this time targeting the site's cookie. This page loads content from several different sources, and contains an XSS vulnerability that we exploit to inject the exfiltration code. Because the underlying framework automatically labels the cookie with the domain of origin, we did not need to insert labeling code. Instead, we find it sufficient to implement a network monitor that checks only whether data sent to an origin does not contain third-party principals. This monitor detected the exfiltration of the cookie (labeled with `ebay.com`) being sent to a server other than `ebay.com`. Again, WebKit's developer tools assisted us with pinpointing the JavaScript code responsible for the request.

8 Related Work

Developer Accessible Labels: To the best of our knowledge, no other work incorporates a first-class labeling system into a dynamically typed programming language. This feature allows the developer to construct label objects, apply them to label other program values, compose them together, and use them as part of natively programmed policy functions.

Myers et al. [15] introduce a security-type system that allows annotation of Java types with confidentiality labels that refer to variables of the dependent type `label` [16]. Java does not represent types as first-class entities, but the Jif programmer does have the ability to use the labeling features to program functions with statically type-checked information flow properties. Our work provides a similar, but simpler, labeling system for the dynamically-typed JavaScript.

Li and Zdancewic [12] present a security sublanguage that expresses and enforces information-flow policies in Haskell. Their implementation supports dynamic security lattices, run-time code privileges, and declassification without modifications to Haskell itself. The type-checking proceeds in two stages: (1) checking and compilation of the base language followed by (2) checking of the secure computations at runtime just prior to execution of programs written in the sublanguage. In contrast, our work presents extensions to an existing JavaScript environment and does not require rewriting of existing programs into a secure sublanguage.

JavaScript Information Flow Systems: Other research on language-based information flow specific to JavaScript relies on automatic labeling frameworks that seek to provide end-users with secure browsers and minimize developer. Our system seeks to leverage web developer domain knowledge about their application as part of a security testing environment.

Vogt et al. [18] modify Firefox's JavaScript engine, SpiderMonkey, to monitor the flow of sensitive information using a combination of static and dynamic analysis. Before execution, their modified VM statically analyzes each function via abstract interpretation to detect and mark implicit information flows. Their framework automatically taints objects provided by the browser (e.g., Document, History, Window, and form elements) and enforces information flows according to the Same Origin Policy. Our supporting framework also automatically labels dynamically loaded code according to the Same Origin Policy, but our contribution of first-class labels allows the developer to specify security policies specific to their application in native JavaScript.

Just et al. [10] modify the JavaScriptCore VM in WebKit to perform information flow tracking for `eval`, `break`, `continue`, and other control-flow structures. Our supporting framework achieves the same analysis with better performance due to difference in implementation details. This work moves beyond implementation of an information flow tracking engine to reflect portions of the labeling engine into the JavaScript environment, to enable targeted security debugging.

Chugh et al. [4] attack the problem of dynamically loaded JavaScript by using staged information flow. Their approach statically computes an information flow graph for all available code, leaving *holes* where code might appear at runtime, and subjecting dynamically loaded code to the same analysis as soon as it becomes available. They also introduce a new policy language to the existing babel of languages used for web development. In contrast, our supporting framework avoids delaying code execution and shifts analysis of information flows to runtime and enables the developer to write policies in JavaScript itself.

Jang et al. [9] employ a JavaScript rewriting-based information flow engine to document 43 cases of history sniffing within the Alexa [1] Global Top 50,000 sites. In contrast, our supporting framework performs label propagation in the VM, increasing performance and preventing attackers from subverting the system.

Type-Checking JavaScript for Information Flow: Many researchers give type systems intended to analyze JavaScript programs for information flow secu-

rity. Austin and Flannagan, in conjunction with Mozilla, promote sparse labeling techniques intended to decrease memory overhead and increase performance [2] and provide a formal semantics for *partially leaked* information [3]. Hedin and Sabelfeld [7] provide Coq-verified formal rules that cover object semantics, higher-order functions, exceptions, and dynamic code evaluation, powerful enough to support DOM functionality. Efforts along this line of research typically cover a core of the JavaScript specification, and have not seen widespread adoption. We forgo formalized verification in a practical effort to target adoption of our work by developers focused on security debugging rather than end users.

JavaScript Language Policies: Meyerovich and Livshits introduce an aspect oriented framework, named ConScript [13] that supports weaving specific security policies with existing web applications. Using their framework, web authors wrap application code with security monitors specified in JavaScript. Their system supports aspect wrapper functions around arbitrary code, while we focus on monitoring network traffic. An aspect oriented approach cannot detect and prevent information leaks that occur due to control-flow transfers as exhibited in Listing 1.1.

9 Conclusion

We present to the JavaScript developer a first-class labeling system that exposes an underlying information flow framework. Developers can use their domain knowledge to label JavaScript values within their application and construct network monitor policies that selectively ignore automatically applied labels. Our labeling system provides dynamic creation of security principals, supporting the common practice of loading code and resources from many different domains in web applications.

We introduce a new built-in `FlowLabelObject` class, which the developer uses to selectively label JavaScript values. The developer creates `FlowLabelObject` instances using existing JavaScript values as security principals or by composition with other `FlowLabelObject` instances via the lattice `join` method. The `subsumes` method allows comparison of all `FlowLabelObject` instances reporting their subset relation within the label lattice. Together with the ability to retrieve labels attached to values via the new built-in `labelof` operator, our system gives the developer the means to implement security policies in JavaScript.

The first-class labeling system introduces no additional slowdown beyond that of an information flow VM, enabling its use in a testing environment for sites that have large amounts of JavaScript code. By leveraging their domain knowledge and existing JavaScript experience, developers can focus on identifying and debugging application specific information flows. First-class labels allow developers to improve the security of their applications by writing policies in JavaScript that selectively ignore the high quantity of reports produced by automatically attached labels.

References

1. Alexa: Alexa Global Top Sites. http://www.alexa.com/topsites (2012), (checked: February, 2013)
2. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 113–124. ACM (2009)
3. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 1–12. ACM (2010)
4. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: PLDI '09: Programming Language Design and Implementation. pp. 50–62. ACM (2009)
5. Denning, D.E.: A lattice model of secure information flow. In: Communications of the ACM. pp. 236–243. ACM (1976)
6. ECMA International: Standard ECMA-262. The ECMAScript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm (2009), (checked: February, 2013)
7. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proceedings of the Computer Security Foundations Symposium. pp. 3–18 (2012)
8. Hennigan, E., Kerschbaumer, C., Brunthaler, S., Franz, M.: Tracking information flow for dynamically typed programming languages by instruction set extension. Tech. rep., University of California Irvine (2011), http://ssllab.org/~nsf/files/tr_instruction_set_extension.pdf
9. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in JavaScript web applications. In: CCS '10: Computer and Communications Security. pp. 270–283. ACM (2010)
10. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information flow analysis for JavaScript. In: PLASTIC '11: Programming Language and Systems Technologies for Internet Clients. pp. 9–18. ACM (2011)
11. K.F., D.P.: XSS Attacks Information. http://www.xssed.com/ (2012), (checked: February, 2013)
12. Li, P., Zdancewic, S.: Encoding information flow in haskell. In: Computer Security Foundations Workshop, 2006. 19th IEEE. pp. 12–pp. IEEE (2006)
13. Meyerovich, L.A., Livshits, B.: ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In: SSP '10: Symposium on Security and Privacy. pp. 481–496 (2010)
14. Mozilla Foundation: Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript (2008), (checked: February, 2013)
15. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. http://www.cs.cornell.edu/jif (2001), (checked: February, 2013)
16. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. In: IEEE Journal on Selected Areas in Communications. pp. 5–19. IEEE (2003)
17. SunSpider: SunSpider JavaScript benchmark. http://www2.webkit.org/perf/sunspider-1.0/sunspider.html (2012), (checked: February, 2013)
18. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: NDSS '07: Network and Distributed System Security Symposium (2007)