A

Information Flow Tracking meets Just-In-Time Compilation

CHRISTOPH KERSCHBAUMER, University of California, Irvine
ERIC HENNIGAN, University of California, Irvine
PER LARSEN, University of California, Irvine
STEFAN BRUNTHALER, University of California, Irvine
MICHAEL FRANZ, University of California, Irvine

Web applications are vulnerable to cross-site scripting attacks that enable data thefts. Information flow tracking in web browsers can prevent communication of sensitive data to unintended recipients and thereby stop such data thefts. Unfortunately, existing solutions have focused on incorporating information flow into browsers' JavaScript interpreters, rather than just-in-time compilers, rendering the resulting performance non-competitive. Few users will switch to a safer browser if it comes at the cost of significantly degrading web application performance.

We present the first information flow tracking JavaScript engine that is based on a true just-in-time compiler, and that thereby outperforms all previous interpreter-based information flow tracking JavaScript engines by more than a factor of two. Our JIT-based engine (i) has the same coverage as previous interpreter based solutions, (ii) requires reasonable implementation effort, and (iii) introduces new optimizations to achieve acceptable performance. When evaluated against three industry standard JavaScript benchmark suites, there is still an average slowdown of 73% over engines that do not support information flow, but this is now well within the range that many users will find an acceptable price for obtaining substantially increased security.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors - Optimization; D.4.6 [**Operating Systems**]: Security and Protection - Information flow controls; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Design, Languages, Security

Additional Key Words and Phrases: JavaScript, Information Flow, Dynamic Language Security

1. MOTIVATION

Dynamically typed programming languages, in particular JavaScript (JS), have become indispensable for powering modern web applications. As the only client-side execution environment supported by all major web browsers, JavaScript has become virtually omnipresent. Today's highly-interactive web applications rely on a performant JavaScript interpreter backed up by a just-in-time compiler.

The browser-hosted execution model permits a web page to dynamically load JavaScript code and other page content in response to user events. Pages using third-party libraries integrate code from different domains within the same execution context. This practice enables feature rich web applications at the same time that it opens a door for attackers to inject code. Malicious code can steal sensitive user information, such as authentication credentials or credit card numbers, without any observable difference in runtime behavior or layout.

Nikiforakis et al. [2012] show that 88.45% of web sites include at least one remote JavaScript library and highlight the potential of such included scripts to perform malicious actions without attracting attention from either web developers or end users. Vulnerability studies consistently rank the code injection attack known as cross-site scripting (XSS) highest in the list of the most prevalent type of attacks on web applications [OWASP 2012; The MITRE Corporation 2012; Microsoft 2012]. A recent empirical study [Jang et al. 2010] of the top 50,000 Alexa sites found that: *popular Web 2.0 applications like mashups, aggregators and sophisticated ad targeting are rife with different kinds of privacy violating flows.*

Web pages often include third-party library code fetched directly by the browser, giving the page administrator no opportunities for screening or sanitizing the included JS code. Even when a site self-hosts all of the application's JS code, user-provided data mixes with the site's pages. Nava and Lindsay [2009] highlight the challenges of server-side analysis techniques and conclude that they can not reliably prevent data injection attacks. Security bugs within a web application provide a path from user-provided data to active JavaScript code executed by subsequent visitors. Consequently, monitoring the flow of information on the client is the most promising strategy to counter unexpected code behavior and prevent information theft attempts.

Several recent approaches [Vogt et al. 2007; Just et al. 2011; Groef et al. 2012; Kerschbaumer et al. 2013] have shown that information flow tracking mitigates the shortcomings of current web security practices and successfully counters XSS-based information theft attacks. Even though these dynamic tracking enhancements provide the desired security, all of the previous approaches suffer from the drawback of incurring performance penalties of at least 80%. Furthermore, they all integrate the tracking logic in the JavaScript interpreter, which itself commonly performs around four times worse than code generated by a just-in-time (JIT) compiler (cf. Section 6.1).

Currently, browser vendors compete for adoption by advertising JavaScript performance. As a result of the "browser wars," faster JavaScript virtual machines (VMs) now enable web applications with large amounts of JavaScript code. Consequently, the slowdown seen when integrating information flow into the JavaScript interpreter represents a major obstacle to adoption. We meet this challenge by implementing dynamic information flow tracking in a JIT compiler. Our framework allows detection of suspicious network traffic that sends data to a server other than that intended by the application programmer. The detection occurs at runtime, catching the code "in flagranti" when performing malicious actions such as data theft.

We first discuss limitations of current web security practices (Section 2), establishing the case for information flow as a viable security technique. Next, we define the different types of information flows (Section 3) and clarify the capabilities of our system. This paper makes the following contributions:

- To the best of our knowledge, we present the first dynamic information flow tracking engine in a JIT compiler for a dynamically typed programming language.
- We present several optimizations (Section 5) that are essential to preserve the performance gains when JIT compiling the information flow tracking logic.
- We evaluate our system (Section 6) along three dimensions:
  - **Efficiency:** We show that our JIT-generated code for information flow tracking introduces, on average, 73% overhead (relative to the unmodified JIT compiler) on established JavaScript benchmark suites; SunSpider, V8, and Kraken, making it 260% faster than the fastest published JavaScript information flow tracking interpreter [Kerschbaumer et al. 2013].
  - **Correctness:** We ensure that our JIT compiler performs accurate information flow tracking by extending Mozilla's regression test suite, demonstrating correct label propagation for binary operations, control-flow structures, `eval`, and function calls.
  - **Applicability:** We implement a web crawler that visits a random sample of 100 web pages from the Alexa Top one million to show that our JIT-generated code for information flow tracking finds the same information flows as our information flow tracking interpreter.

2. THE THREAT OF EXECUTING THIRD PARTY CODE

To date, web applications use the same origin policy (SOP) [Mozilla Foundation 2008] as a first line of defense against information theft. The SOP permits scripts access to methods and properties when sharing the same origin and restricts access otherwise. Unfortunately, rules of the SOP often clash with modern web application architecture, because the SOP only applies to cross-frame communication. For example, mashups intentionally combine third party libraries' web services into a single page and employ code that circumvents the SOP. Kerschbaumer et al. [2013] show that, within the Alexa top 500, some pages contain information influenced by code originating from up to six different domains is sent across domain boundaries. Verification and proof that the mashup performs only the expected task and does not steal data is not available. Hijacking just one commonly included script compromises the privacy of many web users [Nikiforakis et al. 2012].

2.1. The Nature of Code Injection Attacks

Since servers deliver JavaScript as source text, most content and third-party library providers compact their code by automatically shortening variable names, removing all extra whitespace as well as line breaks so the code becomes as small as possible. This practice shortens the transfer time for loading JavaScript, but has the side effect of obfuscating the source code. Web authors are unlikely to audit compressed code for security holes despite the fact that including it in the same execution context as the web application grants it access to application internals. Consequently, a channel through which attackers can inject code that steals sensitive user information remains open and prevalent.

In addition to the risk of including such malicious third party code, we further differentiate between two other forms of code injection attacks. Based on the method of injection, we distinguish between:

- *Non-Persistent (Reflected) attacks*, that occur when client-provided code embedded in HTTP query parameters and HTML form submissions is sent back to the user as page content after processing by the web application servers.
- *Persistent attacks*, that occur when client-provided data is stored on the application's servers and is reflected back to subsequent visitors.

In both of these cases, from the client's perspective, the origin of the attacker code is the same as that of the web application itself, meaning that the SOP can not prevent the attack. Additional security requires a more powerful, behavior-focused mechanism, such as information flow tracking.

2.2. Threat Model

We assume that an attacker has the ability to inject code into a web application. The attacker accomplishes injection by exploiting a XSS vulnerability or the ability to provide content for mashups, advertisements, libraries, etc. which other sites include. To collect the stolen data, we assume that the attacker controls their own web host. The attacker practices only code injection techniques and does not resort to packet sniffing, network interception, or control of the application servers.

2.3. Provided Security

Our modified web browser protects against several information theft attacks, including, but not limited to:

- *Sensitive Data Theft Attacks:* By sending a `GET` request to a server under the attacker's control, the attacker can steal information in the URL of an image request:

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

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

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

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

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

- *Cookie Stealing Attacks:* Furthermore, if a script can access cookies, then an attacker can also steal a session cookie between the browser and an honest site by concatenating the `document.cookie` to the URL of the image request. The stolen cookie allows the attacker to impersonate the user and hijack the user's session.

2.4. Sample Attack: Stealing Form Data

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

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

Listing 1: Example attack code that steals login form data from a web page.

Listing 1 shows exploit code an attacker might use to steal credentials from the login form of a web page. The attack script first loads an image (line 2) supplied by a server under the attacker's control. The attacker designs the image to avoid perceptible changes in page layout. Few users will notice the placement of a single transparent pixel, but the attacker can use the `GET` request as a channel to steal confidential page data whenever the image is reloaded from the server.

The attacker knows users will fill out the form and registers (lines 14, 15) a `blur`-event handler on all forms elements on the page. When a form element loses focus it triggers a call to the `blur`-event handler. The handler, `stealFormData` defined on line 5, first encodes information about the page domain and contents of the form element

which triggered the event in the `payload` variable. Then it updates the `src` attribute of the image with a URL containing the payload. This update causes the browser to automatically reload the image, sending the sensitive information in the URL of the image request.

```
1 [01/Jan/2012:21:34:10] "GET /pixel.png?url=www.bank.com&text=alice HTTP/1.1"
2 [01/Jan/2012:21:34:12] "GET /pixel.png?url=www.bank.com&password=bob69 HTTP/1.1"
```

Listing 2: Log of `attacker.com` from the running example.

By inspecting the server request logs, the attacker can reassemble the captured form data. Listing 2 contains some example entries of image requests. The attacker can clearly identify a user of `www.bank.com` with login ‘`alice`’ having the password ‘`bob69`’.

The webpage *About The Open Web Application Security Project* (OWASP [2013]) hosts an extensive list of XSS vulnerabilities that provides a detailed description of all the different kinds of XSS attacks.

3. TYPES OF INFORMATION FLOWS

Information can flow through a program as a result of either data-flow dependence or control-flow dependence [Denning and Denning 1977]. We examine both of these dependencies to illuminate the ways that an attacker, who manages to craft and inject malicious code, can steal information. The categorization 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 I breaks this category down into two classes: *direct*; corresponding to an immediate dependence; and *indirect*; corresponding to a transitive dependence.

Category	*Class*	*Example*	*Flow*	*Required Analysis*
Explicit	Direct	`b = a`	$a \Rightarrow b$	Dataflow
	Indirect	`b = foo(_, a, _)` `c = bar(_, b, _)`	$a \Rightarrow c$	Dataflow (transitive)

Table I: 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 I 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 II breaks this category down into two classes: *direct*, corresponding to an immediate dependence trackable at runtime; and *indirect*, corresponding to a transitive dependence.

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

Category	*Class*	*Example*	*Flow*	*Required Analysis*

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

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

Table II: Implicit Information Flows.

program counter and a recorded history of control-flow branches taken during program execution. 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. Tracking Capabilities of our Prototype System

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. [Guha et al. 2010] 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:

$$e_1 : l_1 + e_2 : l_2 \hookrightarrow v : l_1 \sqcup l_2 \tag{1}$$

Attackers can also generate implicit flows from confidential to public variables using the control-flow structures in JavaScript [Guha et al. 2010, 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:

$$\textbf{if } (e_{true} : l_{pred}) \; \{ \; e_1 : l_1 \; \} \textbf{ else } \{ \; e_2 : l_2 \; \} \hookrightarrow e_1 : l_{pred} \sqcup l_1 \tag{2}$$

$$\begin{aligned}\textbf{while } (e_1 : l_1) \; \{ \; e_2 : l_2 \; \} \hookrightarrow e_2 : l_1 \sqcup l_2 ; \textbf{ if } (e_1 : l_1) \; \{ \textbf{ while } (e_1 : l_1) \; \{ \; e_2 : l_2 \; \} \; \} \\ \textbf{else } \{ \textbf{ undefined} : \bot \; \}\end{aligned} \tag{3}$$

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 Flana-

gan [2012] gives an example and compares the published mitigation strategies. Unfortunately, we think none of these solutions is a silver bullet. Two of the strategies [Zdancewic 2002; Austin and Flanagan 2010] degrade user experience by halting execution to prevent implicit indirect flows. The third strategy [Vogt et al. 2007] uses a conservative labeling strategy that leads to label creep [Sabelfeld and Myers 2003] in all but trivial cases. Rather than study this design trade-off, we focus solely on the performance impact of JIT compiling the information flow tracking.

4. COMPOSITION OF AN INFORMATION FLOW FRAMEWORK

Whether an interpreter or JIT-compiled code performs information flow tracking, the implementation requires supporting data structures and other modifications to the runtime VM. In this section, we describe the data structures that support multiple security principals necessary for representing the many different domains that can occur on a single web page. We also show how the system encodes labels into a tagged union, allowing propagation of these labels using efficient bit arithmetic.

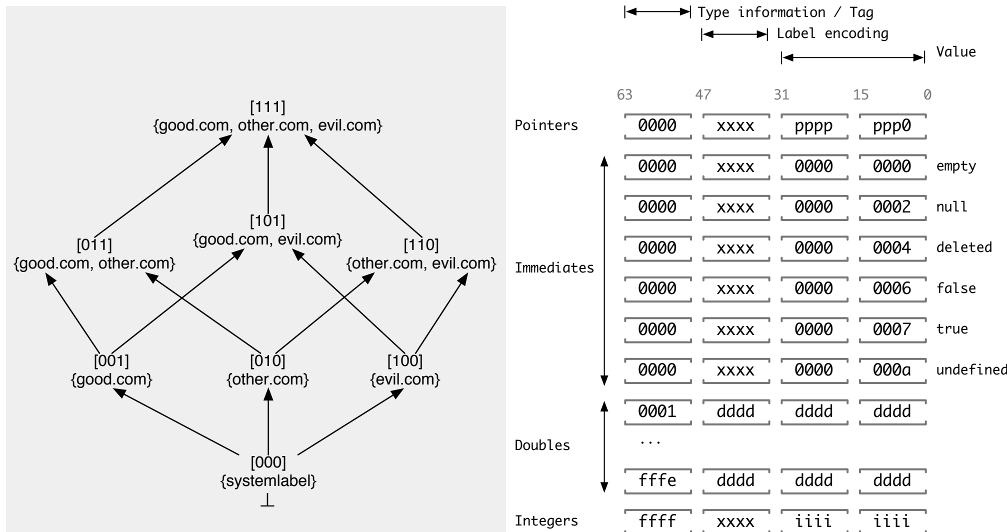

Fig. 1: (left) Label lattice for domains `good.com`, `other.com`, and `evil.com`.
Fig. 1: (right) Label encoding using bits 32-47 in `JSValues`, encoding 16 domains.

4.1. Incorporating a Label Lattice

Within the JavaScript VM, data and objects originating from different domains may interact, creating values that are influenced by multiple domains. To model this behavior, we take inspiration from Myers' decentralized label model [Myers and Liskov 2000] and represent security labels as a lattice join over domains (Figure 1 [left]).

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.

4.2. Encoding Labels in `JSValues`

WebKit already achieves high performance by using a type-tagged union, called `JSValue`, to represent immediate values, object references, and numbers. We repur-

pose 16 of the bits within the `JSValue` representation to hold a security label. This modification allows for a low performance overhead encoding that packs both the label and the typed value within the same 64 bit word.

Pointers/Immediates: `JSValues` starting with the highest 16 bits all set to zero (see Figure 1 [right]) indicate a pointer or immediate type. The VM uses the lowest four bits to distinguish pointers from immediates. Pointers have alignment with these bits all set to zero, while immediate values are non-zero entries in the same lowest four bits: `empty:0x00`, `null:0x02`, `deleted:0x04`, `false:0x06`, `true:0x07`, and `undefined:0x0a`.

In WebKit, addresses occupy 46 bits (bits 0–47). Unfortunately, this design does not leave any space to directly encode a label within `JSValues`. Hence, we modify 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 1 [right]). This 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 gain an 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 [ECMA International 2009] 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 follow the double-precision 64 bit format as specified in the IEEE Standard for Binary Floating-Point arithmetic [IEEE 2008]. 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.

4.3. Tracking Data Flow

Code and data originally tagged with different security principals may interact during execution of a program. The encoding of labels within the lattice supports tagging a single value as having been influenced by multiple principals. For every operation, the information flow VM inspects the labels of all inputs. As formalized in Section 3.3, it constructs a label representing the lattice join over all arguments and the current execution context. The VM then attaches this label to the operation's output value.

For an example of a situation where two principals influence a single value (simplified to omit the current execution context), consider the code snippet `pub += secret;` where the variable `secret` originates from domain `good.com` (`001`) and the variable `pub` originates from domain `evil.com` (`100`). To construct a label that represents this confluence, the runtime VM performs a label join operation (via bitwise or) to obtain the join of the domains `good.com` $\sqcup$ `evil.com` (`001|100`). The updated variable `pub` then carries the resulting label (`101`).

4.4. Tracking Control Flow

Information flow analysis that relies upon static typing (developed for languages such as Jif [Myers et al. 2001]) is not directly applicable to dynamically typed programming languages such as JavaScript. However, we adapt to this situation by implementing a runtime tracking mechanism 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 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 security 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 concurrent static analysis that inserts additional instructions into the sequence. These instructions carry out *pc*-stack operations, maintaining the appropriate stack height and security context label across control-flow joins and branches as the program executes.

4.5. Maintaining the Security Context

Before beginning execution, the JavaScript VM first compiles each function into an instruction sequence. We modify the parser to insert additional instructions that track and record control flow paths executed at runtime. We introduce three new bytecode instructions that serve as convenient markers for control-flow branches and joins in the instruction sequence of a JavaScript function. As illustrated in Figure 2, these instructions perform the required push and pop operations of the *pc*-stack and thus enable runtime control-flow tracking for both the interpreter and the JIT compiler (Section 5.3). We create such a *pc*-stack during setup of each JavaScript stack frame.

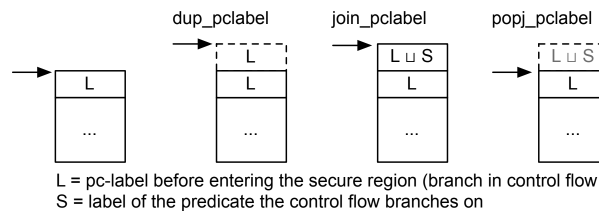

Fig. 2: Maintaining the *pc*-stack using three new instructions.

(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 it 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 pushing on the *pc*-stack from upgrading the top label because loops repeatedly execute the branch condition but retain their control-flow depth. In other words, this design decision avoids unnecessary push operations onto the *pc*-stack and hence improves performance. At least one corresponding `popj_pclabel` instruction later marks the end of the elevated label region.

(2) **`join_pclabel`:** A `join_pclabel` instruction upgrades 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 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 is 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 the 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 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 context 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 (see Listing 3). Loops having more than one exit path (e.g., due to multiple `break` statements) require a `popj_pclabel` at each exit.

4.6. Examining the Security Context

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 3, an attacker can use this feature to construct a correspondence between the induction variable (labeled lower in the security lattice) and a confidential value (labeled higher in the lattice) by breaking out of the loop. Once the loop has terminated early, the attacker returns the induction variable (still labeled lower in the lattice) and leaks the value of the confidential variable.

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

Listing 3: Inferring the value of the variable `secret` by observing the change in control flow using an implicit direct information flow.

JavaScript further complicates the context tracking issue by supporting labeled `break` and `continue` statements that cause early exit from arbitrarily nested inner loops. By performing this action all further operations carried out within the function become tagged with the label under which the break or continue occurred.

We now examine, in greater detail, how the information flow VM inserts instructions into the instruction sequence. Listing 4 contains the instruction sequence for the `stealpin` function shown in Listing 3.

```
[ 0] enter
[ 1] dup_pclabel                              // for (var i=0;   ...    ; ... ) {
[ 2] mov           r0, Int32: 0
[ 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     // for (  ...  ; i < 1000 ; ... )
[31] join_pclabel    r1
[33] loop_if_true    r1, -26(->7)
[36] popj_pclabel    pop:1, join:0            // }
[39] ret             r0                       // return i
```

Listing 4: Instruction sequence of the `stealpin` function in Listing 3.

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 at 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 (offset 7) and gets upgraded (offset 12) after evaluation of the conditional (offset 8). 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 normal exit or by following the `break` statement, the loop ends with a `popj_pclabel` instruction (offset 36) that restores the *pc*-stack to the level it had before loop entry.

4.7. Browser Integration

Solely tracking the flow of information within the JS-engine only provides limited security against data theft attacks. The DOM, for example, provides an interface that allows JS in a web page to reference and modify HTML elements as if they were JS objects. Attacker-supplied JS code can use the DOM as a communication channel for stealing information present in a web page. Our system prevents such data theft attempts by labeling DOM objects based on the origin of their elements and attributes. In this work however, we solely focus on JIT-compiling the information flow tracking logic within the JS-engine and the accompanying performance gain. Kerschbaumer et al. [2013] describes interaction of browser subsystems with the JS-engine.

5. JIT IMPLEMENTATION OF INFORMATION FLOW

Now that we have detailed how the information flow framework encodes labels and dynamically propagates them through both data and control flow at runtime, we present the lower-level implementation which allows the JIT compiler to perform this tracking at substantially improved speeds. Our implementation adds approximately 4,000 lines of C++/assembly code to WebKit's codebase.[1]

5.1. Encoding Labels

When implementing information flow logic in the JIT compiler, native functions impose an additional design constraint. WebKit's JIT compiler and JavaScript interpreter require a unified representation that supports passing `JSValues` between native

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

functions (implemented in C++) and JIT-compiled JavaScript functions. However, the tagged union representation results in high performance, so we find no need to change the label encoding from that introduced in Section 4.2. The label still resides in bits 32–47 of integers, pointers, and immediates, while doubles remain implicitly labeled with the highest available label in the lattice.

5.2. Tracking Data Flow in the JIT

We modify the JIT compiler in WebKit to track information flow in all binary operations: `add`, `sub`, `mul`, `div`, `mod`, `lshift`, `rshift`, `urshift`, `bitand`, `bitor`, and `bitxor`. It is worth noting that JS semantics allow for ad-hoc polymorphism, i.e., using the arithmetic add operator to perform both, numeric addition and string concatenation.

```
1 // join the labels of RAX and RBX
2 MOV R11, RAX // mov first value (including label) into scratch register
3 OR  R11, RBX // bitwise or first value (including label) with second value
4
5 // join the cached top of the label stack
6 MOV R12, [RSP+60h] // load the cached top-label of the pc-stack into scratch register
7 OR  R11, R12       // bitwise or the top-label of the pc-stack with operand labels
8
9 // mask bits, so only label bits remain in R11
10 MOV R12, 0FFFF00000000h // load the label-bit-mask into scratch register
11 AND R11, R12            // bitwise and label-bit-mask with accumulated label
12
13 ADD EBX, EAX // perform the 32 bit add operation
14
15 OR  RBX, R11 // bitwise or the result with the joined label
```

Listing 5: Label propagation for the numeric `add` instruction, with left and right integer operands in registers `RAX` and `RBX` respectively. Registers `R11` and `R12` serve as scratch registers for computing the labels encoded in bits 32–47.

To illustrate how the JIT compiler performs label propagation, we provide a simplified portion of the assembly code emitted by the JIT compiler for integer addition in Listing 5. We split this binary operation into three parts and give a precise description of each computation step:

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

The label on the result of the addition must also include the label from the current context. Because the top of the *pc*-stack provides the security context to every binary operation, the JIT VM caches it in the `JITStackFrame` (Section 5.3). Line 6 retrieves the label from the cache into register `R12`. The VM joins in this context label using another bitwise or, accumulating the result in register `R11` (line 7).

Register `R11` now contains the final label that will be attached to the result of the addition. However, some non-label bits within the register are non-zero, as a result of using the operand values directly. Unfortunately, x86_64 architecture does not support 64 bit immediate operands for the bitwise and operator, so masking out the value bits requires two steps. First, the VM loads a label mask, which has only bits

32–47 set to one, into register `R12` (line 10). Next, the mask and accumulated label undergo bitwise and, leaving only the label's bits active in register `R11` (line 11).

(2) *Performing the Operation:* After calculating the label, the JIT compiler performs the addition of the two operands (line 13) with a 32 bit add instruction. To simplify the example, we elide an overflow check that occurs immediately after the addition. In practice, this check makes use of the overflow and carry flags and transfers control to a slow path that coerces the input integers into doubles.

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

5.3. Optimizing Control Flow Tracking in the JIT

The steps taken to increase JIT compiler performance for tracking control flow involved successive refinement. For example, WebKit's JIT compiler does not fully implement all of the bytecode instructions and often calls back into C++ to handle slow paths. At one stage during development, the JIT compiler implemented our new bytecode instructions (c.f. Section 4.5) through a callback to C++. This stage of implementation had a higher performance overhead than the final product.

Our information flow JIT compiler uses three techniques to enable fast tracking of control-flow influence:

(1) *Our JIT compiler pre-allocates memory for the pc-stack*, just as an unmodified WebKit pre-allocates an array for the call-frame stack. Rather than allocating a small *pc*-stack for each function frame, which negatively impacts runtime performance, each JavaScript function call now reserves space on a global *pc*-stack for the number of labels that it requires for worst-case nesting depth. Reservation of this space is as simple as incrementing a stack pointer in the pre-allocated array, analogous to bump allocation in memory management.

(2) *Our JIT compiler caches the top label of the pc-stack.* Ordinarily, the label of the current execution context is found by following a chain of references that starts at the `CallFrame` pointer in the current `StackFrame`, traverses through the *pc*-stack pointer in the current `CallFrame` and finally ends at an offset from the base of the current *pc*-stack. Because all data-flow operations also join in the current execution context label, the VM caches the top of the *pc*-stack in the `StackFrame` so that it remains accessible through a fixed offset from the `StackFrame`. Figure 3 shows both the chain of references and the cache location.

The JIT compiler updates this cache every time the top label of the *pc*-stack changes. This update occurs at control-flow branches and joins and is less frequent than the data-flow operations that access the top label.

Additionally, using such a caching mechanism allows us to avoid expensive updates on top of the *pc*-stack if the label of the predicate and the cached label are identical. As previously mentioned, the instruction `join_pclabel` upgrades the top of the *pc*-stack by joining it with the label of the predicate value. To avoid such unnecessary updates, our JIT-compiled code first checks if the cached label in the `JITStackFrame`

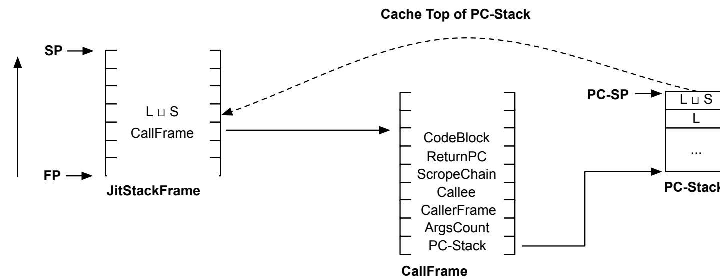

Fig. 3: Interaction of native `JITStackFrame`, `CallFrame`, and *PC*-Stack.

and the label of the predicate are equal. If so, our system skips the expensive task of following the pointers to update the top of the *pc*-stack because the top of the *pc*-stack already holds the correct label.

(3) *Our JIT compiler implements the instructions that maintain the pc-stack directly in assembly.* The JIT compiler takes as input the same instruction stream as the interpreter. When the JIT compiler encounters one of the *pc*-stack manipulation instructions (Section 4.5) it emits assembly code that performs the operation. Not only does this code access the *pc*-stack through the appropriate reference path shown in Figure 3, but it also updates the cached top label when necessary. Cache updates only occur in the `join_pclabel` and `popj_pclabel` instructions, because the `dup_pclabel` instruction modifies stack height but does not change the label on top.

Implementing the instructions that maintain the *pc*-stack in assembly (`dup_pclabel`, `join_pclabel`, `popj_pclabel`) avoids expensive callbacks into C++ at runtime, allowing our JIT-compiler to increase speed by not having to, (i) save and restore registers when calling into C++, and (ii) perform the expensive trampoline jump to find the function entry point in C++.

Only by implementing all of these techniques we were able to achieve the low overhead reported in this paper.

6. EVALUATION

In this section, we evaluate the performance gained by implementing the logic for dynamically tracking information flow in JIT-compiled code. We also emphasize the techniques we use to validate that the implemented logic correctly tracks the flow of information. Finally, we distinguish the limitations that arise as implementation artifacts from the fundamental limitations of the information flow approach.

WebKit contains an interpreter, JavaScriptCore (JSC), that executes a bytecode instruction sequence using direct threaded interpretation[2]. The WebKit project also contains a template JIT compiler that compiles the bytecode stream and an optimizing JIT compiler based on a program's data-flow graph. We have not implemented information flow in the optimizing JIT compiler, because it operates only on Macintosh operating systems. All of the following benchmarks compare information flow implementations of interpreter-only and JIT-only execution modes.

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

Overhead	*Language* (implementation)	*Reference*	*Benchmarks*
80%	JS Interpreter (64 bit labels)	Kerschbaumer et al. [2013]	SunSpider
100 – 200%	JS Interpreter (64 bit labels)	Just et al. [2011]	V8
110 – 690%	JS (rewriting during parse)	Jang et al. [2010]	meas. by visiting pages
120%	JS Interpreter (data-flow only)	Tran et al. [2012]	SunSpider
136 – 560%	JS Interpreter (only tags objects)	Dhawan and Ganapathy [2009]	SunSpider, V8
∼200%	JS Interpreter (multi-execution)	Groef et al. [2012]	V8
none reported	JS Interpreter (1 bit label)	Vogt et al. [2007]	no perf numbers given
14%	Java (data-flow only)	Enck et al. [2010]	CaffeineMark
200%	Java (JikesRVM)	Chandra and Franz [2007]	JavaGrande
1.6% – 26.7%	C (instrumenting compiler)	Nanda et al. [2007]	LAMP-stack
24% – 1,120%	C (instrumenting compiler)	Lam and Chiueh [2006]	C-Programs
1,900%	x86 VM	Yin et al. [2007]	CPU Instruction level tainting

Table III: Performance Comparison of other Information Flow Frameworks

6.1. Effect on Performance

To demonstrate how JIT compilation of dynamic information flow tracking reduces the performance impact within a VM, we execute three established JavaScript benchmark suites: SunSpider version 1.0 [SunSpider 2012], V8 version 6 [Google 2012], and Kraken version 1.1 [Mozilla 2011]. 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) and use `nice -n -20` to minimize operating system scheduler effects. After running each suite once for warm-up, we use 10 repetitions for each benchmark to get stable results and report the geometric mean of these repetitions to discount outliers. Note that the results for our benchmarks do not include the overhead for JIT compiling the code since this happens during the warm-up run.

Previous implementations of information flow (cf. Table III) experience a handicap by beginning with an unmodified interpreter that is already an average of 287% slower than JIT-compiled code. On a relative basis, our system outperforms all of the previous work listed in Table III. Our JIT compiled information flow tracking system also outperforms on an absolute basis because it is measured with respect to much faster JIT-compiled code.

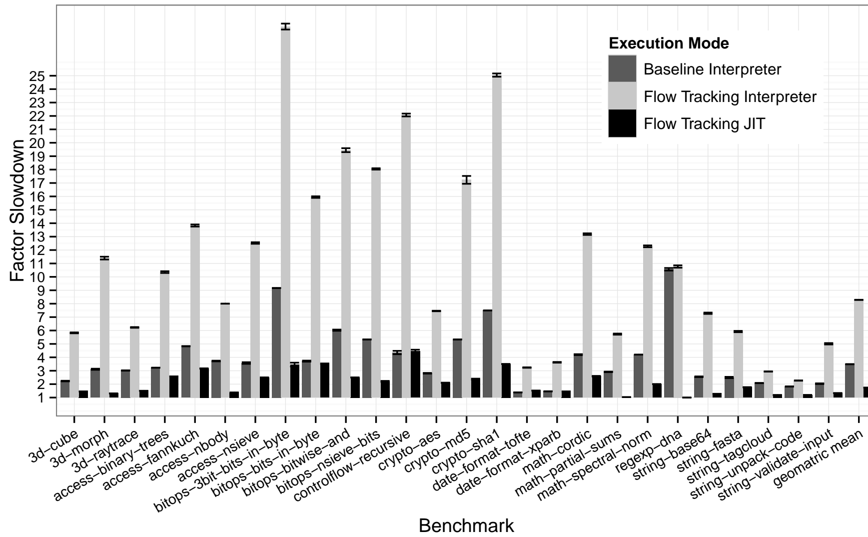

Fig. 4: Detailed performance per SunSpider benchmark normalized by the `JavaScriptCore` JIT compiler.

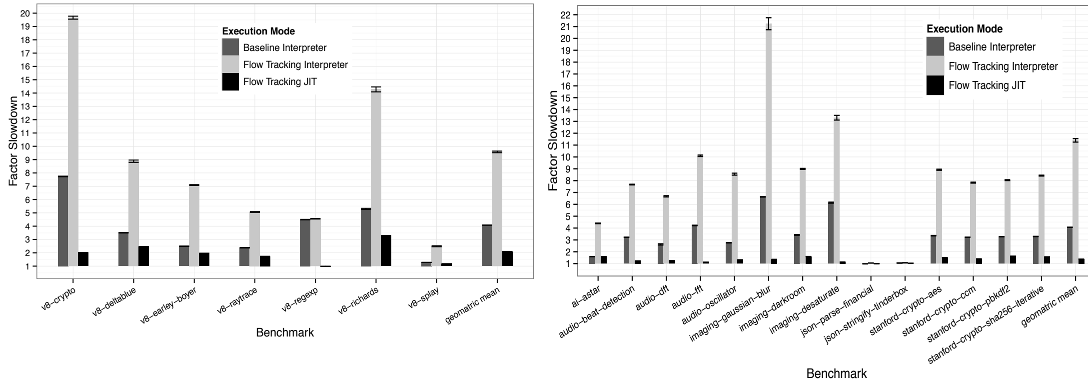

Fig. 5: Detailed performance for V8 (left) and Kraken (right) benchmark normalized by the `JavaScriptCore` JIT compiler.

To provide a consistent basis for performance comparison, we implement information flow tracking in WebKit's interpreter and JIT compiler using the same label encoding and supporting data structures introduced in Section 4. Our tests measure the performance of JIT-compiled tracking code vs. interpreter code in exclusive modes. As illustrated in Figures 4 and 5 the average performance impact for our system (Sunspider 74%, V8 108%, and Kraken 38%) clearly demonstrates that JIT-compiled code implementing dynamic information flow tracking outperforms the execution speed of an unmodified interpreter. Hence, our implementation sets a new bar for dynamic information flow systems.

At the outset, we engineered and incorporated the information flow tracking logic in the JSC interpreter. This effort gave us a deep understanding of WebKit's JavaScript VM and allows us to compare the relative performance impacts of implementing dynamic information flow tracking in the interpreter vs. the JIT compiler. Even when implementing information flow in a JIT compiler, the overhead measured as percentage relative to the baseline does not necessarily correspond to that seen when implementing the same framework within the JavaScript interpreter. For example, the SunSpider benchmark shows an overhead of 137% in the interpreter, while the JIT compiler shows only 74%. A full account of the performance on each individual benchmark (using the encoding and framework as described in this paper) can be found in Appendix A.

Many of the benchmarks, such as `regexp` in V8 and the `json` family of tests in Kraken, run with essentially the same speed as the unmodified JIT compiler. These tests perform fewer control-flow branches and make a higher percentage of native code calls compared to other tests. Meanwhile, the `controlflow-recursive` test in SunSpider introduces the most overhead, at 346%, because it has a very large number of executed branches in recursive function calls and conditional tests. These control-flow constructs cause the dynamic information flow VM to perform additional work to maintain the *pc*-stack. Each time the VM recursively calls a function, branches on a conditional, or iterates a loop, it executes the control-flow tracking instructions (Section 4.5), incurring an overhead relative to an unmodified VM.

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

Impact of Conservatively Labeling Doubles: As previously stated in Section 4.2, the current format of doubles within WebKit does not allow directly encoding a label within the representation of a double. All operations involving doubles implicitly carry the highest label available at the time they execute. This conservative labeling strategy might conceal the performance drawback for benchmarks focusing on doubles.

To show that this implementation detail has little performance impact, we report the percentage of operations creating doubles vs. other `JSValues` for each of the three benchmark suites. In SunSpider 4.7% of `JSValues` created are doubles, while in V8 and Kraken fewer than 1% are doubles, 0.23% in V8 and 0.96% in Kraken, respectively. 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.

6.2. Correctness

To validate that our modifications for tracking the flow of information throughout execution of a JavaScript program do not introduce any errors, we made sure that none of our modifications broke any of the Mozilla regression tests in the WebKit repository. This suite consists of over 1,000 test cases covering core JavaScript functionality, including arrays, dates, functions, numbers, objects, regular expressions, and strings.

In addition, we wrote a suite of test cases that check the correct label propagation for the information flow tracking logic and added them to the regression suite. These tests exercise label propagation for all of the implemented binary operations and control-flow structures: `if`-statements, the various loop constructs including break and continue statements, `eval`, and function calls. Within these tests we make use of a first-class labeling framework [Hennigan et al. 2013] that permits explicit application and inspection of labels within the JavaScript language itself, allowing our tests to be incorporated into the regression suite.

```
1  var a = (new FlowLabel("labelA"))(24);
2  var b = (new FlowLabel("labelB"))(12);
3
4  var res = a + b;
5
6  reportCompare(36, res, "add value incorrect.");
7  reportCompare(true, (labelof res).subsumes(labelof a), "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: Regression test verifying correct label propagation for additions.

Listing 6 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 separate labels. Line 1 assigns input variable `a` the value `24` with label `labelA` (internally mapped to `0001`) and line 2 assigns input variable `b` the value `12` with label `labelB` (internally mapped to `0010`).

After initialization, the test performs the addition on line 4. To provide feedback during development, we use the `reportCompare` function, 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`).

6.3. Real World Applicability

Conforming to the capabilities of the attacker (Section 2), we define an information flow violation as the inequality of domains between a network data payload and the target. When the label of the payload indicates that the data has been influenced by any origin other than the destination domain, the network request represents a communication to a foreign party, possibly an attacker-controlled server.

To verify that our approach detects information flow violations, we implemented a web crawler that automatically visits web pages and stays on each web page for 60 seconds. We randomly sample 100 of the Alexa Top one million [Alexa 2013] web pages for the web crawler to visit. To simulate user interaction, the web crawler fills out HTML-forms and submits the first available form on each visited page. For all of the following results, we ran the crawler using information flow in both the JIT compiler and the interpreter.

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

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

Including other domains: Modern web applications integrate content from several different origins on the web. Our statistics show that each of the visited web pages include an average of 7 different origins for their content. Our approach lets us directly encode up to 16 different domains within one label which allows us to efficiently encode labels even for the web pages including the most content: `prizyvnikmoy.ru`, `finn-dinghy.de`, `mitula.ch`, and `roxio.com` including content from 13 domains. Our findings complement the results of Nikiforakis et al [Nikiforakis et al. 2012], who visited over three million pages for their empirical study showing that more than 90% of all pages include code from less than 15 different domains.

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

The crawler revisited the same pages the following day using the JIT compiler and found 1,173 violations. Because the unit test cases used to develop both the interpreter and JIT compiler implementations attest to the same flow tracking and detection abilities, we attribute the 1.5% variance between runs to dynamic page content. For example, the page `newsarama.com` increased the number of content requests from 11 to 18 between the two runs, where our network monitor recorded 16 violating information flows in the interpreter and 28 information flow violations in the JIT. Groef et al. [2012] report a similar phenomenon when evaluating their system on real web pages.

For our evaluation we do not distinguish between malicious flows and detected flow violations due to the presence of Content Distribution Networks (CDNs), which modern web pages use for performance reasons to serve content to their users. Before our

approach can be adopted, web site authors need a way to express allowed information flows and whitelist requests to their own CDN (cf. Section 6.4).

6.4. Path to Adoption by an Industrial JIT Compiler

Currently, JavaScript VMs do not support any kind of information flow tracking to provide security against information theft attacks. Previous research (cf. Table III) integrates the tracking logic in the JavaScript interpreter, degrading a user's browsing experience. Until now, users desiring this kind of security had to forgo the performance provided by a JIT compiler and execute web applications using an interpreter slowed down by information flow tracking overhead.

JIT compiling the tracking logic allows browsers to execute web applications faster than even an unmodified JavaScript interpreter. By implementing the logic within the JIT compiler, our system provides the same level of protection as other dynamic information flow tracking systems, but with substantially improved performance.

Adopting the Approach: The information flow tracking approach in general still has some remaining challenges. First, conservative labeling leads to a phenomenon known as *label creep* [Sabelfeld and Myers 2003], where labels attached to runtime values (especially those with long lifetimes) steadily rise through the lattice, until most values become labeled with the top element. As the application executes, increasingly more values carry ever higher labels resulting in false positives. We make no attempt to solve this problem and can only suggest research on removing the conservative assumptions through more powerful code analysis.

Second, in the context of JavaScript and the Web, we do not have strong guidelines for what policies to enforce. We expect that most web users will find it too difficult to write their own policies to protect their data. Shipping the browser with a built-in default policy might not be feasible either because web applications vary extensively in both purpose and architecture. Reports on information leakage [Jang et al. 2010; Nikiforakis et al. 2012; Kerschbaumer et al. 2013] suggest that, at this time, most user data is used for web site analytics and marketing. Our tracking engine can be customized to enforce any policy, so we leave questions of policy creation to other researchers, and present here an evaluation that simply counts occurrences of cross-domain network communication.

Finally, as can be seen in the summary in Table III, many information flow systems incur substantial overhead. Our system specifically targets this issue by implementing the dynamic tracking logic in a JIT compiler. Although the percentual slowdown is still similar to other systems, we start from a much faster baseline. By establishing a new status quo for the implementation of information flow, we hope that more users will be willing to adopt these systems.

Adopting the Implementation: The dynamic information flow tracking VM design presented in Section 4 does not implement implicit indirect flow tracking (Section 3.3). Tracking this type of flow requires propagation of control-flow influences through values in non-executed paths and remains an open research question for dynamic languages such as JavaScript.

Our prototype does not cover complete exception handling, and also can not completely handle property lookups. We do not have the engineering power and think that implementing and covering all these features to handle information flow tracking would require expertise from a JS vendor and a dedicated team of engineers.

The repurposing of bits within `JSValues` limits our framework to tracking at most 16 different security principals within a label. We can address this problem in two ways:

- Extend every `JSValue` from 64 to 128 bits, incurring more memory overhead and performance (cf. Table III), but also allows more precise tracking of doubles.

- Reserve one or more bits as a flag to reference a larger label space, requiring a more complex label framework, but offering support for more security principals and a larger lattice (cf. Kerschbaumer et al. [2013]).

We have previous experience implementing both of the these techniques and hold the opinion that the current encoding (Section 4.2) is not a fundamental limitation.

We design the *pc*-stack manipulation instructions (Section 4.5) to support all the different control structures (switch-case, specialized iteration, exceptions, with statement) but did not implement the required instrumentation for any structures beyond the basic for loop in Listing 3. Supporting native data structures (array, string, date, regex, etc.) and property lookup paths (by name, by value, user-overloadable getters/setters, etc.) is required in an industrial strength implementation. Our prototype covers the basic for loop together with mathematical operations, demonstrates that we have an implementation approach with enough shared features between JIT compiled code and the interpreter that it can even support JITing only after the interpreter determines hot code fragments.

We also do not expect an on-the-fly translation between interpreter and JIT-compiled code to be a problem, as long as the trampoline mechanism updates the supporting data structures (label lattice and *pc*-stack) appropriately. Indeed, the introduction of the *pc*-stack maintenance instructions makes this process easier.

7. RELATED WORK

The amount of research on preventing cross-site scripting underscores its importance. In this section, we show how our work relates to the increasingly popular field of information flow security.

Fundamental Information Flow Tracking Systems: Denning and Denning [1977] laid the foundation for subsequent efforts in their seminal work on information flow control. The later survey paper by Sabelfeld and Myers [2003] summarizes research on language-based information flow up until 2003. Most of those efforts focused on static analysis for information flow control in strongly typed languages. For example, Java Information Flow (JIF) [Myers et al. 2001] implements a language-level decentralized label model.

Information Flow Tracking for JavaScript: Unlike statically typed languages such as Java, JavaScript code benefits from dynamic analysis during program execution. JavaScript allows and frequently uses the `eval` function (cf. Nikiforakis et al. [2012]) which converts strings into code. As a result, static analysis techniques can never analyze all code before execution. Unfortunately, dynamic program analysis has drawbacks, too. Unlike static analysis, it both adds to the execution time and restricts analysis to properties of code paths that are actually executed. The latter prevents a single execution of a dynamic analysis from determining implicit indirect flows.

Vogt et al. [2007] pioneered a combination of dynamic data tainting analysis with static analysis inside the Firefox browser. Dhawan and Ganapathy [2009] extended the approach to detect violating flows in browser extensions written in JavaScript. Chugh et al. [2009] separate programs into statically analyzable components and parts that must be dynamically analyzed at runtime. Since the static analysis takes considerable time, it is done at the server side. This has the drawback of requiring cooperation from website operators besides the cost of widespread deployment. Just et al. [2011] improve on Vogt and Dhawan's approaches by improving the analysis of implicit indirect flows to include control dependences created by unstructured control flow.

Several other works on information flow control in JavaScript, such as that by Hedin and Sabelfeld [2012] and Austin and Flanagan [2009; 2010; 2012], influenced the design and implementation of our system. Finally, Kerschbaumer et al. [2013] take inspi-

ration from this research and combine previous approaches into a comprehensive solution that tracks scripting-exposed subsystems in WebKit, including JavaScript VM, the DOM, and user generated events.

We think that the above mentioned approaches can immediately benefit from increased performance via our contribution: information flow tracking for JIT compilers.

Taint Tracking, Secure Multi-Execution, and Isolation: Taint tracking approximates information flow security and is limited to explicit flows. The omission of implicit flows has two advantages: first, the taint tracking overhead is lower since it performs less tracking. Enck et al. [2010], for example, report an average overhead of 14% with their taint tracking solution for Android. Second, full tracking of implicit information flows requires static analysis [Denning and Denning 1977; Myers 1999] or halting execution for some flows [Austin and Flanagan 2009; 2010].

Secure Multi-Execution (SME) is a dynamic execution technique that was developed independently by several researchers. SME prevents all explicit and implicit flows from occurring without the need to handle implicit indirect flows specially, e.g., via static program analysis. Capizzi et al. [2008] multi-execute the entire browser. Devriese and Piessens [2010] formalizes the technique and are able to prove strong soundness and precision guarantees. Recently, Groef et al. [2012] presented FlowFox a full-browser solution that lowers the execution overhead compared to Capizzi et al. by limiting SME to the JavaScript engine. SME unfortunately suffers from high overheads in both time and space. FlowFox, for instance, roughly doubles performance on Google's V8 benchmarks. Austin and Flanagan [2012] use "faceted values" to optimize SME. They also note that a webpage with n principals need up to 2^n executions; FlowFox was evaluated using two.

A number of researchers have evaluated isolation and sandboxing as a defense against XSS and other browser attacks. Grier et al. [2008] built the OP browser which combines formal methods with operating system design principles. It partitions the browser into subsystems with simple interactions and uses information flow to analyze attacks. Nadji et al. [2009] combines randomization of web content with runtime tracking to ensure that untrusted content, included in a page can not be syntactically isolated from its surrounding content. The strength of this approach, called document structure integrity, is its ability to prevent non-JavaScript based XSS attacks; rather than isolating untrusted JavaScript code, some approaches increase JavaScript security by limiting its capabilities. AdSafe, Caja, and FaceBook JS exemplify this approach [Crockford 2009; Miller et al. 2008; Facebook 2011].

Just-In-Time Compilation: Our work leverages the existence of JIT compilers for JavaScript code and is not specific to a particular JIT. Early work on JIT compilation was done by Deutsch and Schiffman [1984] for the Smalltalk-80 system. We refer the interested reader to a concise survey covering the state-of-the-art in JIT compilation until 2003 [Aycock 2003]. Recent advances in JIT compilation for JavaScript were made by Gal et al. [2009] and Hackett and Guo [2012].

8. CONCLUSIONS

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

Our system directly addresses the performance overhead of information flow by implementing the tracking logic in JIT-compiled code. We do not "just" transplant interpretative tracking techniques to a JIT compiler. Rather, we start by carefully choosing a label encoding that is highly efficient w.r.t. space and time. We then (i) optimize

the allocation of the *pc*-stack to track implicit flows, (ii) cache the top of the program counter stack to optimize its accesses, and (iii) inline the code that maintains the *pc*-stack to avoiding costly trampoline jumps. Without these optimizations, a good part of the speedup from JIT compilation would have been lost.

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

ACKNOWLEDGMENTS

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contract No. D11PC20024, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate, Sierra Vista Branch, the National Science Foundation, or any other agency of the U.S. Government.

Thanks to Andrei Homescu for his insightful comments and help to optimize the generated x86 assembly.

REFERENCES

2009. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, June 15-21 (PLDI '09)*. ACM.

2012. *Proceedings of the 19th ACM Conference on Computer and Communications Security, Raleigh, NC, USA, October 16-18 (CCS '12)*. ACM.

2013. *Proceedings of the 6th International Conference on Trust and Trustworthy Computing, London, UK, June 17-19 (TRUST '13)*. Springer.

Alexa. 2013. Alexa Global Top Sites. http://www.alexa.com/topsites. (2013). (checked: April, 2013).

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

Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Toronto, Canada, June 10 (PLAS '10)*. ACM, 1–12.

Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, Philadelphia, PA, USA, January 25-27 (POPL '12)*. ACM, 165–178. DOI:http://dx.doi.org/10.1145/2103656.2103677

John Aycock. 2003. A brief history of just-in-time. *Comput. Surveys* 35, 2 (2003), 97–113. DOI:http://dx.doi.org/10.1145/857076.857077

Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. 2008. Preventing Information Leaks through Shadow Executions. In *Proceedings of the 24th Annual Computer Security Applications Conference, Anaheim, CA, USA, December 8-12 (ACSAC '08)*. IEEE, 322–331.

Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference, Miami Beach, FL, USA, December 10-14 (ACSAC '07)*. IEEE, 463–475. DOI:http://dx.doi.org/10.1109/ACSAC.2007.37

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged Information Flow for JavaScript, See pld [2009], 50–62. DOI:http://dx.doi.org/10.1145/1542476.1542483

Douglas Crockford. 2009. AdSafe. http://www.adsafe.org. (2009). (checked: February, 2013).

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Communications of the ACM* 20, 7 (July 1977), 504–513. DOI:http://dx.doi.org/10.1145/359636.359712

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, Salt Lake City, UT, USA, January (POPL '84)*. ACM, 297–302.

Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 22-25 (SP '10)*. IEEE, 109–124.

Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Proceedings of the 25rd Annual Computer Security Applications Conference, Honolulu, HI, USA, December 7-11 (ACSAC '09)*. IEEE, 382–391. DOI:http://dx.doi.org/10.1109/ACSAC.2009.43

ECMA International. 2009. Standard ECMA-262. The ECMAScript Language Specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm. (2009). (checked: April, 2013).

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Vancouver, BC, October 4-6 (OSDI '10)*. USENIX Association, 393–407.

Facebook. 2011. FBJS (Facebook JavaScript). http://developers.facebook.com/docs/fbjs/. (2011). (checked: February, 2013).

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages, See pld [2009], 465–478. DOI:http://dx.doi.org/10.1145/1542476.1542528

Google. 2012. V8 Benchmark Suite. https://developers.google.com/v8/benchmarks. (2012). (checked: April, 2013).

Chris Grier, Shuo Tang, and Samuel T. King. 2008. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 402–416.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control, See ccs [2012], 748–759. DOI:http://dx.doi.org/10.1145/2382196.2382275

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

Brian Hackett and Shu Guo. 2012. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, China, June 11-16, (PLDI '12)*. ACM, 239–250.

Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow Security for a Core of JavaScript. In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE, 3–18.

Eric Hennigan, Christoph Kerschbaumer, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. First-Class Labels: Using Information Flow to Debug Security Holes, See tru [2013].

IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754–2008* (August 2008), 1–58.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 270–283.

Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. 2011. Information flow analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, Portland, OR, USA, October 22-27 (PLASTIC '11)*. ACM, 9–18. DOI:http://dx.doi.org/10.1145/2093328.2093331

Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Towards Precise and Efficient Information Flow Control in Web Browsers, See tru [2013].

Lap Chung Lam and Tzi-cker Chiueh. 2006. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22rd Annual Computer Security Applications Conference, Miami Beach, FL, USA, December 11-15 (ACSAC '06)*. IEEE, 463–472. DOI:http://dx.doi.org/10.1109/ACSAC.2006.6

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

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

Mozilla. 2011. Kraken JavaScript Benchmark. http://krakenbenchmark.mozilla.org/. (2011). (checked: February, 2013).

Mozilla Foundation. 2008. Same Origin Policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript. (2008). (checked: April, 2013).

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, San Antonio, TX, USA, January 20-22 (POPL '99)*. ACM, 228–241.

Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *Transactions on Software Engineering and Methodology (TOSEM '00)* 9, 4 (October 2000), 410–442.

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. http://www.cs.cornell.edu/jif. (2001). (checked: April, 2013).

Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, Februray 8-11 (NDSS '09)*. The Internet Society.

Susanta Nanda, Lap-Chung Lam, and Tzi-cker Chiueh. 2007. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the ACM / IFIP / USENIX International Conference on Middleware Companion, Newport Beach, CA, USA, November 26-30 (MC '07)*. ACM, 19:1–19:20.

Eduardo Vela Nava and David Lindsay. 2009. 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: April, 2013).

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions, See ccs [2012], 736–747.

OWASP. 2012. The Open Web Application Security Project. https://www.owasp.org/. (2012). (checked: April, 2013).

OWASP. 2013. XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. (2013). (checked: August, 2013).

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. Vol. 21. 5–19.

SunSpider. 2012. SunSpider JavaScript Benchmark. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html. (2012). (checked: April, 2013).

The MITRE Corporation. 2012. Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. http://cwe.mitre.org/top25/. (2012). (checked: April, 2013).

Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. 2012. Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations. In *Proceedings of the 10th International Conference on Applied Cryptography and Network Security, Singapore, June 26-29, (ACNS '12)*. Lecture Notes in Computer Science, Vol. 7341. Springer, 418–435.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, February 28-March 2 (NDSS '07)*. The Internet Society. DOI: http://dx.doi.org/10.1.1.117.6526

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, October 28-31 (CCS '07)*. ACM, 116–127.

Stephan A. Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation.

A. DETAILED BENCHMARK RESULTS

Benchmark	*Base-JIT*	*%*	*Base-Int*	*%*	*Flow-Int*	*%*	*Flow-JIT*	*%*
Kraken-Total	8883.4	(0.0)	36099.4	(306.37)	101244.3	(1039.7)	12266.7	(38.09)
astar	1567.7	(0.0)	2499.4	(59.43)	6893.4	(339.71)	2484.4	(58.47)
beat-detection	648.0	(0.0)	2091.1	(222.7)	4970.4	(667.04)	793.3	(22.42)
dft	652.5	(0.0)	1708.9	(161.9)	4357.2	(567.77)	807.8	(23.8)
fft	482.3	(0.0)	2035.8	(322.1)	4870.5	(909.85)	542.3	(12.44)
oscillator	427.1	(0.0)	1177.4	(175.67)	3650.1	(754.62)	561.4	(31.44)
gaussian-blur	2441.3	(0.0)	16186.3	(563.02)	51855.2	(2024.08)	3298.5	(35.11)
darkroom	701.7	(0.0)	2398.1	(241.76)	6309.1	(799.12)	1116.2	(59.07)
desaturate	691.8	(0.0)	4249.0	(514.19)	9205.9	(1230.72)	785.1	(13.49)
parse-financial	85.1	(0.0)	83.5	(-1.88)	89.4	(5.05)	84.8	(-0.35)
stringify-tinderbox	102.0	(0.0)	107.7	(5.59)	108.9	(6.76)	104.9	(2.84)
crypto-aes	215.6	(0.0)	723.6	(235.62)	1923.0	(791.93)	322.6	(49.63)
crypto-ccm	168.8	(0.0)	544.9	(222.81)	1322.1	(683.23)	235.6	(39.57)
crypto-pbkdf2	533.1	(0.0)	1746.4	(227.59)	4287.2	(704.2)	869.3	(63.07)
crypto-sha256-iterative	166.4	(0.0)	547.3	(228.91)	1401.9	(742.49)	260.5	(56.55)
SunSpider-Total	233.7	(0.0)	815.0	(248.74)	1936.2	(728.5)	406.6	(73.98)
cube	12.6	(0.0)	28.1	(123.02)	73.4	(482.54)	18.3	(45.24)
morph	10.0	(0.0)	31.1	(211.0)	114.0	(1040.0)	13.1	(31.0)
raytrace	11.3	(0.0)	34.1	(201.77)	70.4	(523.01)	17.0	(50.44)
binary-trees	3.1	(0.0)	10.0	(222.58)	32.1	(935.48)	8.0	(158.06)
fannkuch	14.1	(0.0)	68.1	(382.98)	195.0	(1282.98)	44.6	(216.31)
nbody	8.0	(0.0)	29.8	(272.5)	64.0	(700.0)	11.0	(37.5)
nsieve	4.0	(0.0)	14.3	(257.5)	50.1	(1152.5)	10.0	(150.0)
3bit-bits-in-byte	2.4	(0.0)	22.0	(816.67)	68.8	(2766.67)	8.3	(245.83)
bits-in-byte	6.0	(0.0)	22.3	(271.67)	95.7	(1495.0)	21.1	(251.67)
bitwise-and	4.0	(0.0)	24.1	(502.5)	77.8	(1845.0)	10.0	(150.0)
nsieve-bits	6.0	(0.0)	32.0	(433.33)	108.3	(1705.0)	13.2	(120.0)
recursive	2.8	(0.0)	12.2	(335.71)	61.8	(2107.14)	12.5	(346.43)
aes	9.0	(0.0)	25.3	(181.11)	67.1	(645.56)	18.9	(110.0)
md5	3.0	(0.0)	16.0	(433.33)	51.7	(1623.33)	7.1	(136.67)
sha1	2.0	(0.0)	15.0	(650.0)	50.1	(2405.0)	7.0	(250.0)
format-tofte	15.1	(0.0)	20.9	(38.41)	48.9	(223.84)	22.9	(51.66)
format-xparb	11.0	(0.0)	16.1	(46.36)	39.9	(262.73)	16.0	(45.45)
cordic	8.0	(0.0)	33.6	(320.0)	105.5	(1218.75)	20.7	(158.75)
partial-sums	13.0	(0.0)	37.9	(191.54)	74.5	(473.08)	13.4	(3.08)
spectral-norm	5.0	(0.0)	21.0	(320.0)	61.4	(1128.0)	10.0	(100.0)
dna	15.0	(0.0)	158.5	(956.67)	161.6	(977.33)	15.0	(0.0)
base64	8.0	(0.0)	20.4	(155.0)	58.3	(628.75)	10.2	(27.5)
fasta	9.1	(0.0)	22.7	(149.45)	53.9	(492.31)	16.1	(76.92)
tagcloud	16.0	(0.0)	33.3	(108.12)	47.1	(194.38)	19.0	(18.75)
unpack-code	26.1	(0.0)	47.7	(82.76)	59.2	(126.82)	31.1	(19.16)
validate-input	9.1	(0.0)	18.5	(103.3)	45.6	(401.1)	12.1	(32.97)
V8-Total	1644.4	(0.0)	6706.4	(307.83)	15754.7	(858.08)	3426.7	(108.39)
crypto	239.9	(0.0)	1856.2	(673.74)	4716.4	(1865.99)	482.9	(101.29)
deltablue	378.7	(0.0)	1326.8	(250.36)	3362.9	(788.01)	932.1	(146.13)
earley-boyer	171.6	(0.0)	427.6	(149.18)	1216.5	(608.92)	336.5	(96.1)
raytrace	105.0	(0.0)	249.8	(137.9)	531.8	(406.48)	181.5	(72.86)
regexp	201.0	(0.0)	903.3	(349.4)	916.1	(355.77)	197.7	(-1.64)
richards	309.3	(0.0)	1637.6	(429.45)	4415.4	(1327.55)	1015.4	(228.29)
splay	238.9	(0.0)	305.1	(27.71)	595.6	(149.31)	280.6	(17.46)

Table V: Detailed performance numbers for Kraken, Sunspider, and V8 benchmarks normalized by the JavaScriptCore JIT compiler.