CrowdFlow: Efficient Information Flow Security

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

University of California, Irvine

Abstract. The widespread use of JavaScript (JS) as the dominant web programming language opens the door to attacks such as Cross Site Scripting that steal sensitive information from users. Information flow tracking successfully addresses current browser security shortcomings, but current implementations incur a significant runtime overhead cost that prevents adoption.
We present a novel approach to information flow security that distributes the tracking workload across all page visitors by probabilistically switching between two JavaScript execution modes. Our framework reports attempts to steal information from a user's browser to a third party that maintains a blacklist of malicious URLs. Participating users can then benefit from receiving warnings about blacklisted URLs, similar to anti-phishing filters.
Our measurements indicate that our approach is both *efficient* and *effective*. First, our technique is efficient because it reduces performance impact by an order of magnitude. Second, our system is effective, i.e. , it detects 99.45% of all information flow violations on the Alexa Top 500 pages using a conservative 5% sampling rate. Most sites need fewer samples in practice; and will therefore incur even less overhead.

1 Motivation

Modern web pages have become complex web applications that mash up scripts from different origins inside a single execution context in a user's browser. 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 type of attacks on web applications [1,2,3]. Attackers use XSS to gain access to confidential user information. A recent study on privacy violating flows confirms the ubiquity of user data theft when browsing the web [4].

Previous work on browser security shows that information flow tracking can counter such attacks [5,6,7,8,9]. Even though information flow tracking prevents misappropriation of sensitive data, all known approaches introduce runtime overheads that make execution of JS code at least two to three times slower. We believe that industry will never adopt the information flow approach without a substantial reduction in this overhead.

Taint tracking is a more efficiently implementable subset of information flow tracking; for example, TaintDroid [10] reports an overhead of just 14%. Information flow tracking increases security by tracking both data and control flow, but unfortunately no efficient implementation is known for dynamically typed languages such as JS.

Our solution distributes the tracking overhead among a crowd of visitors, leveraging the same property that attackers target: site popularity. 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 applications much faster than traditional approaches where every client always performs the costly information flow tracking. Even though the CrowdFlow approach permits individuals to miss detection of specific information flow violations, we show that a crowd of users, in aggregate, detects the majority of information flow violations. Clients report policy violating flows to a trusted third party that collects suspicious information flow reports, similar to commercial blacklisting initiatives like Google's *Safe Browsing* [11] or Microsoft's *Smartscreen-Filter* [12].

Currently, corporations hosting URL blacklist services populate the database at their own expense, through automated scanning that tends to miss real-world use of web applications by logged-in users. These services also provide a form through which end-users can submit a malicious URL for investigation, but this collection mechanism tends to catch code that causes user-level annoyance rather than surreptitious and silent data theft. Additionally, website operators in adversarial competition submit false allegations in an attempt to put competing websites on the blacklist.

We believe that automating the reporting process on the client side and basing it on privacy-violating information flow results in three benefits. First, automated reporting increases the amount of data that these systems have, enabling them to improve report validation. Second, automated reporting reduces the number of false allegations by raising the bar on the level of detail a report contains. Third, automated reporting tracks into the deep web, inspecting application behavior after a user has logged in. CrowdFlow, with its low per-user overhead, is a perfect front-end for these systems.

We provide background information on JS security (Section 2) that motivates the development of CrowdFlow, define the threat model our system defends against (Section 3) and make the following contributions:

- We introduce CrowdFlow (Section 4), a novel approach to information flow tracking that switches between two JS interpreters to balance performance and security. This architecture distributes the tracking costs across a crowd of visitors to a page.
- We present a comprehensive information flow tracking browser (Section 5) based on WebKit [13] and provide implementation details for both partial taint tracking and information flow tracking modes.
- We evaluate our system on a variety of real-world websites. In particular, we demonstrate the practicality of our framework (Section 6) by showing that our system satisfies the following important properties:
  - **Efficiency:** CrowdFlow executes JS an order of magnitude faster than traditional approaches for information flow tracking, with an average runtime overhead of 27.84% for SunSpider [14] and 32.05% for V8 [15] benchmarks. To compare, execution overhead of traditional information flow implementations ranges between 200% and 300%.

– **Effectiveness:** Our approach finds almost all (99.45%) information flow violations on the Alexa Top 500 [16] web sites compared to a traditional information flow tracking system. We achieve this detection rate with a crowd of only five users, and a conservative function invocation sampling rate of 5%.

2 Background on JS Security

XSS is a code injection attack that allows an adversary to execute code without the user's knowledge and consent. For example, XSS allows attackers to harvest sensitive information such as keystrokes, authentication credentials and credit card numbers. A malicious script can even traverse the Document Object Model (DOM) [17] and steal all visible data on a compromised web page [18].

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. [19] 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.

Currently, browsers rely on the Same Origin Policy (SOP) [20], and the Content Security Policy (CSP) [21] to limit a script's access to information. The CSP allows page authors to whitelist trusted sources and the SOP prevents access for scripts of different origins when properly isolated with `iframe`-tags. However, neither policy can prevent JS from stealing information on a page when developers include multiple libraries in the same execution context, as currently practiced [19].

3 Threat Model

Throughout this paper we assume that attackers have the following abilities: (i) attackers can operate their own hosts, and (ii) 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., that victim sites include. The attacker's capabilities, however, are limited to JS and the attacker can neither intercept nor control network traffic.

Phishing Campaigns vs. Targeted Attacks: In contrast to common information flow tracking systems, the architecture of CrowdFlow does not attempt to prevent information theft attacks from within the user's browser. Rather, it reports detected information flow violations to a trusted third-party aggregator, such as Google's *Safebrowsing* initiative or Microsoft's *Smartscreen-Filter*. CrowdFlow is not designed to defend against a targeted attack, in which the attacker tries to steal information of one particular person. The architecture of CrowdFlow aims to protect the majority of users against phishing campaigns, where the attacker distributes exploit code to high-traffic web pages to gather as much information as possible.

Threat Examples CrowdFlow Defends Against: To steal information from a browser, malicious code must surreptitiously communicate it to an attacker-controlled server. For example, by placing an image on the page, the attacker can steal sensitive information through the target URL of an image request:

```
1 elem.src = "evil.com/p.png?v=" + creditcard_number;
```

The `GET` request for the image `p.png` acts as a channel through which the attacker steals the user's credit card number as a query parameter of the target URL. The attacker-controlled server records the image request, including the stolen data, in its logs.

4 CrowdFlow

The design of traditional JS information flow tracking systems requires every client to track all information flows [5,6,7,8,9]. In contrast, CrowdFlow implements a probabilistic approach, where each user only spends a fraction of the execution time in the slower information flow tracking interpreter, thus paying only a fraction of the performance cost. Following the distributed system design of the Internet itself, CrowdFlow distributes the security analysis across a crowd of visitors, aggregates the flow reports at a trusted third party, and shares findings back to users, warning them of potentially malicious pages.

4.1 Probabilistic Tracking

The CrowdFlow browser primarily executes in a partial taint tracking interpreter (state PTT in Figure 1) that propagates labels only across direct assignments (`a = b;`).

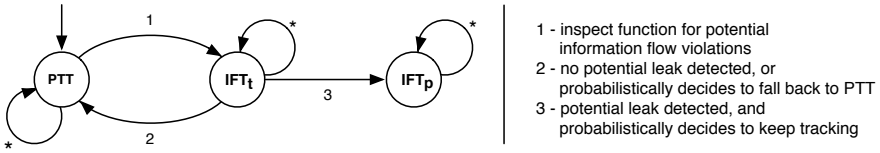

Fig. 1: Execution states in CrowdFlow. PTT - Partial Taint Tracking, IFT$_t$ - Information Flow Tracking (trial), IFT$_p$ - Information Flow Tracking (permanent).

CrowdFlow has a configurable *sampling rate*, that controls the switch from state PTT to the *trial* information flow tracking interpreter (state IFT) at every function invocation. Both IFT modes propagate every operation's dependence on control-flow predicates (see Section 5.2), preventing malicious code from using inference of control-flow branches to circumvent the partial taint tracking. When executing in IFT$_t$ mode, CrowdFlow watches for operations that involve the mixing of data from multiple domains, as this occurrence indicates a potential information flow violation.

Definition of a Potential Information Flow Violation: *We define a potential information flow violation as the result of two domains influencing a value.*

For example, assume variable `a` originates from domain `A` and variable `b` originates from domain `B`, then `b += a;` constitutes a potential information flow violation because

data from both domains `A` and `B` influence the resulting value of variable `b`. When malicious code attempts to steal data from a page, the copy or encoding operations involved follow this definition and CrowdFlow detects the confluence of values from multiple domains.

When no potential violation occurs in the *trial* information flow tracking mode (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 IFT_t, it probabilistically switches to the *permanent* information flow tracking interpreter (state IFT_p). The probability of transferring to state IFT_p and continue tracking 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.

4.2 Tracking Multiple Domains

Our system tracks the flow of information throughout program execution by applying a label to every program value. These labels take the form of a bit-vector that encodes information about a program's origin (Section 5.4). CrowdFlow maintains a registry of all domains represented on a page, mapping a unique bit to each page. When running in information flow tracking mode, CrowdFlow labels each value resulting from an operation with the set union of all domains of all inputs, including implicit inputs such as the predicates of any currently executing branches and the origin of the code itself.

4.3 Reporting Information Flows

CrowdFlow tracks flows of information not only in the JS engine, but also across scripting-exposed browser subsystems, including the DOM and user-generated events. During execution, CrowdFlow monitors network traffic for information leaks.

Definition of an Information Leak: *We define an information leak 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. CrowdFlow detects the attempt and reports the source domains involved in the leak and the target URL to a commercial blacklisting initiative. We use the defined *Example Threat* from our Threat Model, as the running example to explain how CrowdFlow detects such an information leak.

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

Using a XSS vulnerability the attacker injects the example code on a page from host `bank.com`. When loading the page, the CrowdFlow browser maps the host URL `bank.com` to a unique label bit, say `0001`. Because this snippet appears within the page, it has access to all the host page's application content.

To steal information, the malicious code appends the sensitive information stored in host variable `creditcard_number` as part of the target query-string for an image

request (line 1). Setting the source attribute of an image element on the host page causes the browser to issue a `GET` request to `evil.com`. CrowdFlow registers the new domain with another unique label bit, `0010`. Before emitting the request on the network, CrowdFlow inspects the label on the payload (`0001`) and finds that it differs from the target (`0010`), triggering an information flow violation report.

Note that the same code, even when dynamically loaded from `evil.com`, also triggers a flow report. In this case, the malicious code carries label of `evil.com` while the host variable `creditcard_number` still carries the label of `bank.com`. As a result of CrowdFlow's label propagation rules, the computed `url` payload carries the join of these domains (`0011`), which differs from the target domain (`0010`).

5 Implementation

A single web page can incorporate data from several different domains. Within the JS engine, data and objects originating from different domains (security principals) may interact, creating values that derive from more than one domain. To model this behavior, we take inspiration from Myers' decentralized label model [22] and represent security labels as a lattice join over domains. Internally, the CrowdFlow browser associates each domain with a unique marker and implements joins as a set union over domains.

5.1 Partial Taint Tracking Interpreter

We implement the CrowdFlow browser by modifying WebKit, which ships with a register-based direct-threaded JS interpreter (`JavaScriptCore`), so that all values carry a label indicating the domains that influenced its construction. The partial taint tracking interpreter operates on tainted data and efficiently propagates labels for direct assignments due to our label encoding: Because the label resides in the virtual machine level representation of a JS value, a direct assignment from one variable to another also carries that label, without additional computation logic.

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

This assignment shows that the content of `pub` depends directly on the value 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 has the value of variable `secret`, but it also carries the label of variable `secret`, since the assignment is a full copy of the variable contents. Again, the partial taint tracking interpreter propagates labels only for direct assignments.

5.2 Information Flow Tracking Interpreter

Conventional static analysis techniques for information flow, such as those developed for the Java-based Jif [23], are not directly applicable to dynamically typed languages, such as JS. However, we adapt these techniques by introducing a *control-flow stack* that manages labels for different regions of a running program, which is a common technique for securing programs [5,9]. At runtime, CrowdFlow updates the label on top of this

stack at every control-flow branch and join within a program, to model entry and exit points for secure regions of a program. The top of the control-flow stack always contains the current security label of the current *program counter*, which carries the set join of predicates in all enclosing branches.

Tracking Data Flow: The following accumulation operator shows the content of variable `secret` adding or concatenating with the public variable `pub`.

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

This code snippet illustrates how CrowdFlow can stop a specific data theft attempt. An attacker gathers sensitive information on a web page, but before the attacker can steal 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. The information flow tracking interpreter tracks the operation by joining the labels of the operands of the addition/concatenation together with the label of the current program counter.

Tracking Control Flow: The following code snippet shows an implicit *direct* information flow [24] which occurs when a control-flow branch predicate influences a value.

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

As illustrated, the script code 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. The CrowdFlow information flow tracking interpreter propagates implicit direct information flows by updating the label of the current program counter to reflect its dependence on the variable `secret`. At the assignment (line 3), the variable `pub` becomes tainted with the label of `secret` by virtue of joining with the current program counter.

The efficient handling of implicit *indirect* information flows [24], where information can be inferred by inspecting values in the non-executed path, still remains an open research question. Our implementation can not track such implicit indirect information flows. The browser information flow system presented by Vogt et al. [5] for example, jumps to a conservative secure mode if their static analysis detects a function call or use `eval` in the non-executed branch. CrowdFlow does not implement this technique because it steadily elevates labels on all values and objects, leading to a phenomenon known as *label creep* [25].

5.3 Switching Interpreters

The naive way to implement our technique adds a condition 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 an information flow tracking instruction set in addition to the existing instruction set. We make efficient use of WebKit's

direct-threaded JS interpreter by duplicating opcodes and providing an information flow tracking equivalent implementation of every opcode.

For example, the opcode `op_add` now also has an information flow tracking equivalent `op_ift_add`. 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. Having two instruction streams allows fast and easy switching between the partial taint tracking and the information flow tracking interpreter by directing the interpreter's instruction pointer to either the original or our modified information flow tracking instruction stream at function entry.

5.4 Multi-Domain Label encoding

We implement security labeling by repurposing the memory layout of `JSValues`, the virtual machine level representation of a JS value in WebKit. This modification of bits inside `JSValues` allows for low overhead encoding of a 16-bit label within the 64-bit word size indicating the origin.

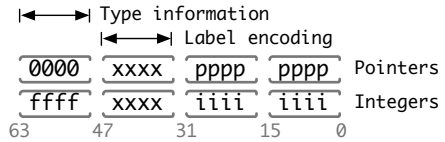

Fig. 2: Label encoding using bits 32–47 in `JSValues`.

Pointers/Immediates: `JSValues` starting with the highest 16 bits all set to zero (see Figure 2), indicate a pointer or immediate type. Pointers have alignment that forces the lowest four bits to be zero. This encoding allows WebKit to efficiently distinguish real pointers from immediate values which are all encoded in the lowest four bits: `null:0x02`, `false:0x06`, `true:0x07`, `undefined:0x0a`.

The actual address of a pointer in WebKit uses 48 bits (bits 0–47). This design unfortunately does not leave any space to directly encode a label for pointers within `JSValues`. To encode a label, we repurpose bits and change the current encoding of pointers. We use `mmap` with the `32_BIT` flag, to force memory allocations to be within the 32 bit address space, freeing up 16 bits (bits 32–47) in the pointer address space. Using these 16 bits allows us to encode up to 16 different domains in a label (marked as `xxxx`).

Kerschbaumer et al. [9] show that web pages, on average, include content from 12 different domains. They also provide a technique for overcoming the space limitation for encoding domains in values by reserving the highest bit as an overflow flag, indicating that the page includes content from more domains than the available encoding space, where the lower bits become an index into an array. Furthermore, this design of encoding labels allows us to use efficient bit arithmetic for label join operations that propagate labels within the browser and equality operations that detect information leaks at network requests.

Integers/Doubles: Values starting with the highest 16 bits all set to one indicate an integer type. The ECMAScript specification [26] defines JS integers to be 31-bit. To encode security labels in integers we can also make use of the bits 32–47, which are unused, even in the original WebKit encoding of `JSValues`.

WebKit's encoding reserves all other values (highest 16 bits between `0x0001` and `0xfffe`) for doubles. Since doubles in JS follow the double-precision 64 bit format, there are no bits left for tagging `JSValue` doubles. Therefore we conservatively label doubles by using the highest currently available security label in the lattice (i.e., the join of all registered domains).

6 Evaluation

6.1 Security (Effectiveness)

To measure how well CrowdFlow matches the capabilities of a traditional information flow tracking system, we simulate a crowd of users with a web crawler that automatically visits the Alexa Top 500 web pages and stays on each web page for 60 seconds. The crawler simulates user interaction by filling out and submitting the first available HTML-form on each visited page.

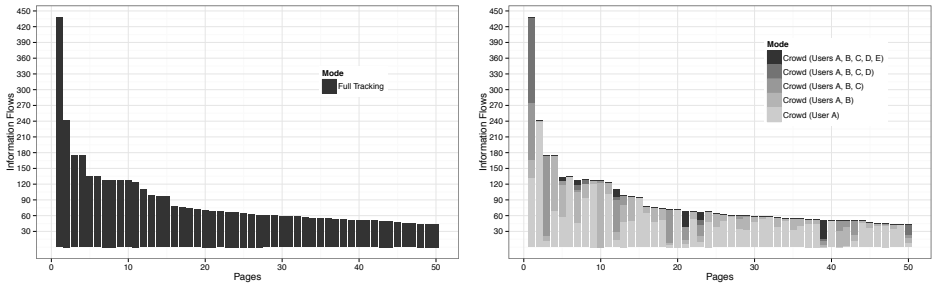

Fig. 3: Reported information flow violations for the 50 pages that trigger the most warnings when visiting the Alexa Top 500 pages. One user executing the information flow tracking interpreter vs. a crowd of up to 5 users using CrowdFlow.

Full (Baseline) Information Flow Tracking: To establish a baseline against which to compare CrowdFlow we arrange for the crawler to run in permanent information flow tracking mode (state IFT_p in Figure 1). This experiment detected information flows across domain boundaries on 433 of the Alexa Top 500 pages. The crawler detected a total of 8,764 such flows which are sent to a total of 1,384 distinct domains on the Internet. Together, the Alexa Top 500 pages use a total of 391,930 different JS functions (as of 2012/12/24) which are invoked 13.5 million times in total.

CrowdFlow: To show that the detection rate provided by CrowdFlow converges with that of traditional information flow tracking systems, we revisit the Alexa Top 500 pages using CrowdFlow and compare the results against the baseline. To evaluate this claim

we set CrowdFlow's sampling rate at 5%. For popular sites, this setting "oversamples" given the number of visitors seen in practice. However, we chose this rate because it allows evaluation of CrowdFlow with a small, crawler-simulated crowd of five users.

Figure 3 (left) 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 flow 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 3 (left) 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 3 (right) shows the detected information flows by five CrowdFlow clients when revisiting the 50 pages having the most information flows on the Alexa Top 500 pages. 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 3) 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 3) 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.

In total, the crawler-simulated crowd of five visitors found 8,716 information flows out of 8,764 (4,357 out of 4,359 in Figure 3) reported by a traditional information flow tracking system, which represents a detection rate of 99.45%.

6.2 Performance (Efficiency)

To evaluate how CrowdFlow reduces the performance penalty of information flow tracking within browsers, we modified 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 evaluating the performance impact of our framework, we measure performance using the SunSpider [14] and the V8 [15] benchmark suites. Both are frequently used to evaluate JS security and therefore facilitate comparison to related work.

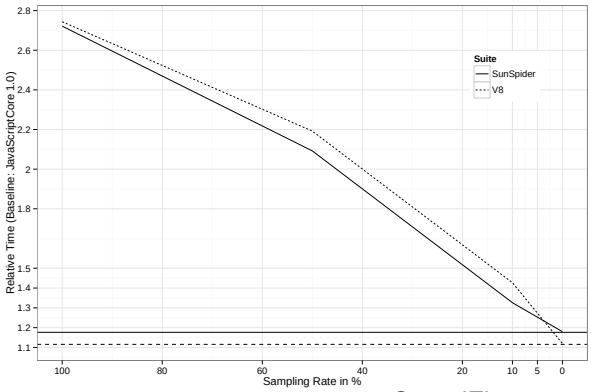

Fig. 4: Performance impact of CrowdFlow.

Figure 4 shows 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 five percent sampling rate reduces this overhead by 5×, down to about 30% overhead compared to `JavaScriptCore`. The lower, horizontal 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 the 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 results in almost ten percent further performance improvement.

6.3 Discussion and Limitations

Currently browsers do not support any kind of information flow tracking and provide little security against information theft attacks. Previous information flow tracking systems support only full tracking which severely affects a user's browsing experience. CrowdFlow provides a balanced, flexible approach that trades the guarantee of 100% information flow tracking in return for improved performance. In aggregate, the CrowdFlow approach captures almost all of the information flows found by the full tracking system, but at a much lower per-user performance cost.

Approach limitations: Our multi-domain labeling strategy allows our system to clearly identify Content Distribution Networks (CDNs) which modern web pages use for performance reasons to serve content to their users. Before our approach can be adopted, we need a policy that allows web site authors to express allowed information flows, for example, flows within their own CDNs (cf. [27]). For example, a declaration of such a policy in the HTTP header, similar to the approach of Jim et al. [28], is feasible. At the moment, we also leave statistical analysis of the information flow reports up to a third-party aggregator (commercial URL blacklisting service).

Implementation limitations: Dynamic information flow tracking systems are susceptible to timing channel attacks, and ours is no exception. At this time we are primarily concerned with passive adversaries, those that are not actively trying to subvert our countermeasures. Therefore, we consider this problem out-of-scope and are focused on improving the speed of tracking. 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 [29] and Zdancewic [30].

A privacy-violating flow report may reveal information about the user who reported it. In the current implementation, CrowdFlow elides all information about the state of the web application and restricts the report contents to contain only the set of source domains and the target domain involved in the privacy-violating flow. To hide information from the URL blacklisting service about who visited what site, we can also incorporate a traffic anonymizing service such as TOR [31].

7 Related Work

Distributed Dataflow Analysis: In 2011, Greathouse et al. [32,33] 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.

Information Flow Systems: The survey paper of Sabelfeld and Myers [25] puts the related work in the area of language-based information flow up until 2003 into perspective.

In 2007, Vogt et al. [5] present their implementation of information flow control in the Firefox browser. In 2010, Russo et al. [34] provide a mechanism for tracking information flow within dynamic tree structures. In 2011, Just et al. [7] present their information flow system, improving upon results made by Vogt et al. Finally, in 2012 De Groef et al. [6] describe their implementation of secure-multi-execution [35] in the Firefox browser to give strong information flow security guarantees.

CrowdFlow shares similarities and takes inspirations 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.

There exist many other approaches to secure JavaScript, such as previous work by Hedin and Sabelfeld [36], Austin and Flanagan [37,34,8], Chugh et al. [38], and Nadji et al. [39]. The key differentiator between these approaches and CrowdFlow is practicality. Our system has an efficient implementation and does not require invasive changes to the existing web architecture.

Third-Party Security Systems: In 2011, Canali et al. present a system called Prophiler [40] and Thomas et al. present a system called Monarch [41]. Both approaches describe details of machine learning techniques used to classify malware on the web.

For CrowdFlow, both of these projects (and the commercial blacklisting initiatives mentioned previously) are complimentary because our approach adds efficient and effective information flow tracking as another source of input. For example, the analysis performed by Prophiler or the rich honey-clients used in Monarch can prioritize URLs better with data from CrowdFlow reports.

8 Conclusion

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 both performant code execution by participating clients and prevention of attacker code from deterministically 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 across the crowd of visitors to a web page.

CrowdFlow can report privacy-violating information flows to a blacklisting URL service. Users benefit from their participation in information flow tracking by receiving warnings about malicious code on a page.

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 5% function invocation sampling rate.

Acknowledgements

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 Michael Bebenita, Stephen Crane, Andrei Homescu, Christopher Horn, Mark Murphy, Mathias Payer, Codrut Stancu, Gregor Wagner, Christian Wimmer, and Wei Zhang for their feedback and insightful comments.

References

1. OWASP: The open web application security project. `https://www.owasp.org/` (2012) (checked: April, 2013).
2. The MITRE Corporation: Common weakness enumeration: A community-developed dictionary of software weakness types. `http://cwe.mitre.org/top25/` (2012) (checked: April, 2013).
3. Microsoft: Microsoft Security Intelligence Report, Volume 13: January - June 2012. `http://www.microsoft.com/security/sir/default.aspx` (2012) (checked: April, 2013).
4. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in JavaScript web applications. In: Proceedings of the ACM Conference on Computer and Communications Security, ACM (2010) 270–283
5. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: 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)
6. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a web browser with flexible and precise information flow control. In: Proceedings of the ACM Conference on Computer and Communications Security, ACM (2012) 748–759
7. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information flow analysis for JavaScript. In: Proceedings of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, ACM (2011) 9–18
8. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, ACM (2012) 165–178

9. Kerschbaumer, C., Hennigan, E., Larsen, P., Brunthaler, S., Franz, M.: Towards precise and efficient information flow control in web browsers. [42]
10. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. (2010) 393–407
11. Provos, N.: 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: April, 2013).
12. Microsoft: SmartScreen Filter. http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter (2012) (checked: April, 2013).
13. WebKit: The webkit open source project. http://www.webkit.org (2012) (checked: April, 2013).
14. SunSpider: SunSpider JavaScript benchmark. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html (2012) (checked: April, 2013).
15. Google: V8 Benchmark Suite. https://developers.google.com/v8/benchmarks (2013) (checked: April, 2013).
16. Alexa: Alexa Global Top Sites. (http://www.alexa.com/topsites) (checked: April, 2013).
17. W3C - World Wide Web Consortium: Document object model (DOM) level 3 core specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf (2004) (checked: April, 2013).
18. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking information flow in dynamic tree structures. In: Proceedings of the European Symposium on Research in Computer Security, Springer (2009) 86–103
19. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S.V., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: Large-scale evaluation of remote javascript inclusions. In: Proceedings of the ACM Conference on Computer and Communications Security, ACM (2012) 736–747
20. Mozilla Foundation: Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript (2008) (checked: April, 2013).
21. W3C: Content security policy 1.0. http://www.w3.org/TR/CSP/ (2013) (checked: July, 2013).
22. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology **9** (2000) 410–442
23. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. http://www.cs.cornell.edu/jif (2001) (checked: April, 2013).
24. Hennigan, E., Kerschbaumer, C., Larsen, P., Brunthaler, S., Franz, M.: First-class labels: Using information flow to debug security holes. [42]
25. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Selected Areas in IEEE Communications **21** (2003) 5–19
26. Ecma International: Standard ECMA-262. The ECMAScript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm (2009) (checked: April, 2013).
27. Anonymous: Web statistics when crawling the alexa top 500 web pages. Technical report, Anonymous (2013)
28. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: Proceedings of the ACM International Conference on World Wide Web, ACM (2007)

29. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages, ACM (1999) 228–241
30. Zdancewic, S.A.: Programming Languages for information security. PhD thesis, Cornell University (2002)
31. The Tor Project: Tor: Anonymity Online. `https://www.torproject.org/` (2013) (checked: April, 2013).
32. Greathouse, J.L., LeBlanc, C., Austin, T., Bertacco, V.: Highly scalable distributed dataflow analysis. In: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, IEEE (2011) 277–288
33. Greathouse, J.L., Austin, T.: The potential of sampling for dynamic analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, ACM (2011) 3:1–3:6
34. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, ACM (2010) 1–12
35. Devriese, D., Peissens, F.: Noninterference through secure multi-execution. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE (2010) 109–124
36. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proceedings of the IEEE Computer Security Foundations Symposium, IEEE (2012) 3–18
37. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, ACM (2009) 113–124
38. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM (2009) 50–62
39. Nadji, Y., Saxena, P., Song, D.: 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)
40. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: A fast filter for the large-scale detection of malicious web pages. In: Proceedings of the ACM International Conference on World Wide Web, ACM (2011) 197–206
41. Thomas, K., Grie, C., Ma, J., Paxson, V., Song, D.: Design and evaluation of a real-time url spam filtering service. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE (2011) 447–462
42. Proceedings of the 6th International Conference on Trust and Trustworthy Computing, London, UK, June 17-19 (TRUST '13), Springer (2013)

A Detailed Benchmark Results

Benchmark	*JSCore (%)*	*PTT %*	*IFT %*	*Crowd %*
V8-Total	6691.5 (0.0)	7466.8 (11.59)	18362.1 (174.41)	8835.9 (32.05)
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)
Sunspider-Total	807.6 (0.0)	950.2 (17.66)	2198.0 (172.16)	1032.4 (27.84)
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)
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)
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)
recursive	12.0 (0.0)	17.0 (41.67)	70.2 (485.0)	21.6 (80.0)
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)
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)
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)
dna	159.5 (0.0)	158.2 (-0.82)	159.5 (0.0)	159.9 (0.25)
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)

Table 1: Detailed performance numbers for V8 and Sunspider benchmarks normalized by the JavaScriptCore interpreter.