# Reactive non-interference for a Browser Model

Nataliia Bielova*, Dominique Devriese†, Fabio Massacci* and Frank Piessens†
* University of Trento, Italy
{bielova, massacci}@disi.unitn.it
† DistriNet Research Group, KULeuven, Belgium
{dominique.devriese, frank.piessens}@cs.kuleuven.be

***Abstract*—We investigate non-interference (secure information flow) policies for web browsers, replacing or complementing the *Same Origin Policy*. First, we adapt a recently proposed dynamic information flow enforcement mechanism to support asynchronous I/O. We prove detailed security and precision results for this enforcement mechanism, and implement it for the Featherweight Firefox browser model. Second, we investigate three useful web browser security policies that can be enforced by our mechanism, and demonstrate their value and limitations.**

## I. Introduction

The explosive growth of Web applications such as web-based e-mail, social networking, web banking, and others has turned the Web into one of the most important software delivery platforms. The Web browser has become a virtual machine that receives and executes a variety of interactive applications from different stakeholders. Hence, one of the key security responsibilities of a browser is to provide proper protection mechanisms to ensure that these different applications cannot interfere with each other in non-authorized ways. In today's browsers, this is achieved by enforcing the *same-origin-policy*. An *origin* is a (protocol, domain name, port) triple, and restrictions are imposed on the way in which code and data from different origins can interact.

Unfortunately, this same-origin-policy is fraught with problems. Not only is it implemented inconsistently in current browsers [1], it is also ambiguous and imprecise [2], and it fails to provide adequate protection for resources belonging to the user rather than to some origin [1]. This has led to a significant amount of research proposing improvements for web browser security, ranging from specific countermeasures for holes in the same-origin-policy to proposals for new browser architectures that basically turn a browser into a service operating system. We give a brief overview of this research area in the related work section.

Of particular importance for this paper are the various proposals that have been made to base the policy enforced by a browser on *non-interference*, or *information flow security*. A program is non-interferent if secret inputs to the program do not influence (*flow into*) public outputs. An information flow policy defines which inputs and outputs are considered secret or public. More generally, a policy has a partially ordered set (poset) of security or confidentiality levels and labels input and output channels with such levels. The program is non-interferent if information only flows from inputs labeled $l_i$ to outputs labeled $l_o$ for $l_i \leq l_o$. In other words: information only flows upward, toward more confidential levels.

Non-interference has been studied intensely for several decades, and a wide variety of enforcement mechanisms has been proposed. Sabelfeld and Myers [3] provide an extensive survey of static enforcement methods, and Le Guernic [4] surveys dynamic methods. Several authors have already investigated the use of secure information flow techniques in the context of a browser, for instance to secure mashup composition [5], [6], or to prevent private information to flow to advertisement providers [7], [8].

Devriese and Piessens [9] introduced a novel *secure multi-execution* technique that proposes to execute a program multiple times, once for each security level, using special rules for I/O operations. The main advantage of this approach is that it is proved to be sound (any program is noninterferent under secure multi-execution) and precise (the technique does not change the behaviour of the noninterferent programs).

Very recently, Bohannon et al. [10] proposed to replace the same-origin-policy with information flow policies. They define the notion of *reactive non-interference*, an adaptation of the classic notion of non-interference to *reactive systems*, systems that perform asynchronous I/O such as web browsers. In addition, they provide a bisimulation-based proof technique to prove the soundness of enforcement mechanisms for reactive non-interference. In a later paper, Bohannon et al. [2] develop Featherweight Firefox, an extensive formalization of a web browser as a reactive system (implemented in OCaml).

Continuing on this line of work, this paper makes the following contributions. First, we develop a provably secure and precise enforcement mechanism for reactive non-interference, based on secure multi-execution [9]. Our precision results are stronger and more general than those proven for the original technique. Then, we analyze three useful web browser policies, of increasing complexity. We demonstrate their value and limitations in some typical web scenario's. Finally, we implement our mechanism and all the policies for the Featherweight Firefox browser model.

The remainder of this paper is structured as follows: The

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven, and by the EU FP7 programmes: EU-ICT-IP-MASTER, EU-FET-IP-SecureChange, FP7-ICT-WebSand and FP7-IST-NoE-NESSOS. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). The authors wish to thank Aaron Bohannon for his helpful comments on an earlier version of this paper.

next section describes the problem addressed in this paper. We summarize the results of Bohannon et al. used in the this paper in Section III. Section IV gives an informal overview of our approach and Section V provides a formal model where we prove our main precision and security results. Section VI presents a variety of useful policies that can be enforced by our mechanism. Finally, we provide related work, and conclude.

## II. Problem statement

A browser interacts with a variety of web sites, and possibly executes JavaScript code downloaded from them. In order to make sure that these sites do not interfere in undesirable ways, today's browsers enforce the same-origin policy, an access-control policy where browser resources are tagged with their origin, and access to resources is limited to code coming from the same origin. Origin is defined as a triple (protocol, host, port), so two origins are considered to be the same only if all the elements of their tuples are equal.

The same-origin-policy has many problems, and has been criticized by many authors [1], [11]. Some of the issues, such as the fact that different browser resources use different definitions of origin, can be considered implementation bugs or inconsistencies, and they could in principle be addressed without fundamentally changing the same-origin access control policy (even though, as Singh et al. point out [1], the incompatibility burden of such fixes can be substantial). While such issues are important, they are not what this paper is about.

Other limitations of the same-origin-policy are more fundamental, and do not seem to be solvable without significant changes to the policy enforced by the browser. In particular, there are several scenarios that indicate that a policy based on non-interference would have advantages over the current access control policy. A first, very simple, motivating example is a scenario where a website sends code to perform calculations on user private data.

*Example 1 (Tax Calculator):* Suppose the fictitious website `http://taxcalc.com` offers the service of pre-calculating the amount of tax in function of income, age, marital status and so forth. The service sends an HTML form for entering the user's information, and JavaScript code for calculating the tax based on the information entered in the form.

The user wants assurance that the information he enters does not leave his computer — not even to the website providing the service. The same-origin-policy does not offer any protection for this scenario since the origin of the script for calculating the tax is the same as the origin of the including page.

More fundamentally, if we assume that further interactions between the user and the website are essential (for instance to pay for the service), no access-control policy can provide this assurance: the script needs access to the private data to perform its function, and it needs access to the network to send invoicing information to the service. What is needed is an information flow enforcement mechanism that can ensure that the script cannot leak private information to the network.

*Example 2 (Flight ticket):* Consider an e-commerce site where users can order flight tickets. Obviously, the user will be fine with sharing some private information such as name, birth date and even credit-card information with the website. However, the user would like to have assurance that this information does not leak to other sites.

The same-origin-policy provides some protection for this scenario: it ensures that scripts running in the user's browser and belonging to web pages from other origins cannot access the information entered by the user. However, scripts that are part of the e-commerce web pages will have the same origin, so they can access and easily transmit information to other sites. This can be done by initiating an HTTP request to that other site where some information to be leaked is encoded in the URL or parameters of the request [12]. The script that leaks the information does not necessarily come from the trusted site, there are many ways in which malicious scripts can find their way into pages from trusted websites. Two common attack vectors are (1) cross-site scripting (XSS), where a vulnerability in the server software enables an attacker to inject scripts in the web pages served by the server [13], and (2) the inclusion of advertisements from third-party ad-providers; such advertisements are regularly implemented as scripts that run within the same origin as the including page [7].

An important additional challenge is that for many web applications, some form of information flow between origins is actually desired. So any proposed browser security policy should not block such information flows. It is, for instance, common to include content (e.g. images and scripts) from other origins in web pages. A strict non-interference policy would prohibit such techniques and hence be strongly incompatible with the current web.

The examples above illustrate that non-interference is a promising candidate for a (baseline) browser security policy, but two important problems need to be addressed.

First, an enforcement mechanism for non-interference at the level of the browser is needed. While several browser security countermeasures based on information flow security techniques have been proposed, none of them can enforce non-interference for the full browser and for a broad class of security lattices in a secure and precise way (see the Related Work section). This paper proposes an enforcement mechanism, and proves it secure and precise.

Second, non-interference is parameterized with a policy: a poset of security levels, and an assignment of such levels to browser inputs and outputs. Selecting suitable policies is a challenge. This paper analyzes several interesting policies and shows that they can securely handle the scenarios above, yet stay compatible with desired cross-origin information flows such as image and script loading.

## III. Background

To address the first problem (the development of a general, secure and precise enforcement mechanism for a full browser), we need a formal model of a browser and a formalization of non-interference for such a model. This section summarizes work by Bohannan et al. on Featherweight Firefox [2] and reactive non-interference [10] that we build on in this paper.

### A. Reactive systems

At the highest level of abstraction, a browser is modeled as a *reactive system* [2], [10], a particular kind of automaton that reacts to inputs by changing state and emitting outputs.

*Definition 3.1:* A *reactive system* is a tuple

$$(\mathit{ConsumerState}, \mathit{ProducerState}, \mathit{Input}, \mathit{Output}, \rightarrow)$$

where $\rightarrow$ is a labelled transition system whose states are $\mathit{State} = \mathit{ConsumerState} \cup \mathit{ProducerState}$ and whose labels are $\mathit{Act} = \mathit{Input} \cup \mathit{Ouput}$, subject to the constraints:

- for all $C \in \mathit{ConsumerState}$, if $C \xrightarrow{a} Q$, then $a \in \mathit{Input}$ and $Q \in \mathit{ProducerState}$,
- for all $P \in \mathit{ProducerState}$, if $P \xrightarrow{a} Q$, then $a \in \mathit{Output}$,
- for all $C \in \mathit{ConsumerState}$ and $i \in \mathit{Input}$, there exists a $P \in \mathit{ProducerState}$ such that $C \xrightarrow{i} P$, and
- for all $P \in \mathit{ProducerState}$, there exists an $o \in \mathit{Output}$ and $Q \in \mathit{State}$ such that $P \xrightarrow{o} Q$.

The system is idle and is waiting for inputs in consumer states, and it emits outputs in producer states. A reactive system can only handle one input event at a time (thus correctly modeling the fact that JavaScript event handlers are single threaded). The definition allows for non-termination: it is possible that the system never returns to a consumer state. We limit our attention in this paper to deterministic reactive systems.

Reactive systems transform streams of input events into streams of output events. A *stream* is defined as a coinductive interpretation of the grammar $S ::= [] \mid s :: S$, where $s$ ranges over stream elements. A coinductive definition of the grammar defines the set of finite and infinite objects that can be built with repeated applications of the term constructors, so a stream is a finite or infinite list of elements [14].

We use metavariables $I$ and $O$ to range over streams of inputs $i$ and outputs $o$, respectively. The *behavior* of a reactive system in a state $Q$ is defined as a relation between the input streams and output streams.

*Definition 3.2:* Coinductively define $Q(I) \Rightarrow O$ (state $Q$ transforms the input stream $I$ to the output stream $O$) by the following rules, where $C$ and $P$ are respectively consumer and producer states: $C([]) \Rightarrow []$

$$\frac{C \xrightarrow{i} P \qquad P(I) \Rightarrow O}{C(i :: I) \Rightarrow O} \qquad\qquad \frac{P \xrightarrow{o} Q \qquad Q(I) \Rightarrow O}{P(I) \Rightarrow o :: O}$$

### B. Featherweight Firefox

The notion of reactive system is very abstract. To analyze potential security policies, we use a browser model that concretizes the abstract states, inputs and outputs. The *Featherweight Firefox* browser model [2] does exactly that. It includes many browser features such as multiple browser windows; cookies; sending HTTP requests and receiving HTTP responses; essential HTML elements; building document node trees, and also the basic features of JavaScript. It is implemented as an executable model in OCaml.

Featherweight Firefox (FF) is a reactive system, with a much more detailed definition of the input and output events, and the internal state of the browser. Input events can either come from the user (loading a URL in a new window `load_in_new_window`, entering text in a text box `input_text`, etc.), or from the network (receiving an HTTP response `receive`). Output events can be to the user (web page is updated `page_updated`, window is opened `window_opened`) or to the network (sending HTTP request `send`). The FF browser model defines precisely how the browser will react to these inputs by emitting outputs. Some of input and output events are shown in Table I.

TABLE I: Selected user and network I/O events.

| | |
|---|---|
| User input | `load_in_new_window`(url)<br>`input_text`(user_window, nat, string) |
| User output | `window_opened`<br>`page_loaded`(user_window, url, rendered_doc)<br>`page_updated`(user_window, rendered_doc) |
| Network input | `receive`(domain, nat, cookie_updates, resp_body) |
| Network output | `send`(domain, request_uri, cookies, string) |

The FF model is surprisingly rich. We will see examples including for instance the execution of event handlers implemented as scripts in an html page.

### C. ID-security, or reactive non-interference

It remains to define what it means for a reactive system (and hence FF) to be non-interferent. Bohannon et al. [10] propose a notion of *ID-security*, a termination insensitive variant of non-interference. We specialize their definitions to this case.

Let us assume that a poset of security levels is given. The predicate $\mathit{visible}_l(s)$ models what observers of security level $l$ can see: $\mathit{visible}_l(s)$ is true iff the stream element $s$ is visible to an observer at level $l$. First, we define what it means for two (input or output) streams to be equivalent up to level $l$.

*Definition 3.3:* Coinductively define $S \approx_l^{ID} S'$ ($S$ is ID-similar to $S'$ at $l$) with the following rules:

$$[] \approx_l^{ID} [] \qquad\qquad \frac{\mathit{visible}_l(s) \qquad S \approx_l^{ID} S'}{s :: S \approx_l^{ID} s :: S'}$$

$$\frac{\neg \mathit{visible}_l(s) \qquad S \approx_l^{ID} S'}{s :: S \approx_l^{ID} S'} \qquad\qquad \frac{\neg \mathit{visible}_l(s) \qquad S \approx_l^{ID} S'}{S \approx_l^{ID} s :: S'}$$

This definition is coinductive, meaning that the property holds on the largest possible set fixed under all the rules. We can now define when a reactive system is secure in a state $Q$.

*Definition 3.4:* A state $Q$ is *ID-secure* or *(reactive) non-interferent* if, for all $l$, $I \approx_l^{ID} I'$ implies $O \approx_l^{ID} O'$ whenever $Q(I) \Rightarrow O$ and $Q(I') \Rightarrow O'$.

The definitions in this section allow us to state our first goal for this paper: we want to build an enforcement mechanism ensuring that FF is reactive non-interferent.

## IV. Informal overview

Our enforcement mechanism is based on a relatively new dynamic technique for achieving non-interference: secure multi-execution [9], [15], [16]. The idea of this mechanism is to execute the program multiple times (one sub-execution of the program for each security level), and to ensure that (1)

```
var a = parseInt(document
   .getElementById('a').value);
var b = parseInt(document
   .getElementById('b').value);
var sum = a + b;
document.getElementById('c').value = sum;
document.getElementById('banner')
   .src = 'http://attacker.com?t=' + sum;
```

Fig. 1: JavaScript code example

outputs of a given level $l$ are only done in the sub-execution at level $l$, and (2) inputs at level $l$ are only done at level $l$ (the sub-executions above $l$ reuse the inputs obtained by level $l$; sub-executions not above $l$ are fed a default value). So the sub-execution at level $l$ only sees inputs of levels below $l$ and its output could not have been influenced by inputs of a higher level. Non-interference follows easily from this observation.

Devriese and Piessens [9] have worked out this mechanism for a simple sequential programming language with synchronous I/O, and have proven its security and precision. Capizzi et al. [15] have implemented it at the level of operating system processes for the case of two security levels.

The mechanism we propose adapts this technique to reactive systems, and we prove its security (weaker than what Devriese and Piessens have shown in their setting: we lose termination- and timing-sensitivity), as well as its precision (stronger than the result by Devriese and Piessens: we show precision under much weaker assumptions).

Let us explain the mechanism by means of an example. Consider again the tax calculation example from Section II. The JavaScript code in Fig. 1 models the essence of this example: the user provides private inputs (two integers) in the text fields $a$ and $b$, and the JavaScript code computes their sum and displays this in text field $c$. We can assume this JavaScript code is a part of an event handler that fires whenever the user changes the contents of $a$ or $b$.

The code in Fig. 1 shows a potential attack: the script will leak the (secret) sum to ”attacker.com” by sending an HTTP request to that domain with the secret as a parameter (setting the `src` property of an image HTML element in JavaScript will have as a side effect that the image is reloaded from the URL assigned to the `src` property). Recall that the JavaScript code was not necessarily endorsed by the tax calculation site. It could have been injected through a cross-site scripting (XSS) attack or hidden in an advertisement running on the page. Under a policy that assigns a high security level (H) to text field inputs, and a low security level (L) to all the outputs (including the one to ”attacker.com”), this program is clearly not secure: high inputs leak to low outputs. Notice that all current browsers are vulnerable to such attack since a script gets assigned the same origin as the including page and hence is able to leak any (secret) user information.

Our enforcement mechanism runs several sub-executions of the web browser, one for each security level. Tables II and III show what happens at L and H sub-executions. The level of every input event is shown in column 1, while levels of output events are shown in column 2. The tables show which events get suppressed. For instance, for the L sub-execution, the following events get suppressed: (1) the input events of level H (and also all output events that would have been the result of the input event), and (2) the output events at level H.

TABLE II: Run of L sub-execution of the browser.

| | | |
|---|---|---|
| L | | load_in_new_window(”http://taxcalc.com”) |
| | H | ~~window_opened~~ |
| | L | send(”taxcalc.com”, request_uri, cookies, ...) |
| L | | receive(”taxcalc.com”, 0, cookie_updates, doc(a=0, b=0, c=0, js_inline)) |
| | H | ~~page_loaded(user_window(“taxcalc.com”), ..., doc(a=0, b=0, c=0, js_inline))~~ |
| H | | ~~input_text(user_window(“taxcalc.com”), 1, ”2”)~~ |
| L | | (further L input) |
| | L | send(”attacker.com”, request_uri, cookies, ”?t=0”) |

TABLE III: Run of H sub-execution of the browser.

| | | |
|---|---|---|
| L | | load_in_new_window(”http://taxcalc.com”) |
| | H | window_opened |
| | L | ~~send(”taxcalc.com”, request_uri, cookies, ...))~~ |
| L | | receive(”taxcalc.com”, 0, cookie_updates, doc(a=0, b=0, c=0, js_inline)) |
| | H | page_loaded(user_window(“taxcalc.com”), ... doc(a=0, b=0, c=0, js_inline)) |
| H | | input_text(user_window(“taxcalc.com”), 1, ”2”) |
| | H | page_updated(user_window(“taxcalc.com”), doc(a=0, b=2, c=2, ...)) |
| | H | window_opened |
| | L | ~~send(”attacker.com”, request_uri, cookies, ”?t=2”)~~ |
| L | | (further L input) |
| | L | ~~send(”attacker.com”, request_uri, cookies, ”?t=2”)~~ |

The offending output to ”attacker.com” is suppressed, as the L sub-execution never gets the H input event where the user is typing secret data in the text box. In the tables, we show that even if the script would try to send the contents of $a$ and $b$ later in response to further L input, the actual output sent to “attacker.com” would only contain the sum of the default values in both text boxes. Notice the proposed mechanism also ensures that there are no implicit flows, since the low sub-execution (that is allowed to send) does not contain any high-level information (the user secret data). There is never *any* information flow from H inputs to L outputs.

## V. Formalization

We propose to apply the secure multi-execution technique to a reactive system. Given an information flow policy, we build a new reactive system that is called a *wrapper*. The wrapper runs multiple *sub-executions* of the original reactive system: one for each security level. When it consumes an input event, it is passed to those sub-executions that are allowed to see it, i.e. the sub-executions at a level higher or equal than level of this event. A sub-execution produces output events only at the level of this sub-execution. Because of space constraints, proofs are provided in a separate technical report [17].

### A. Secure multi-execution of reactive systems

The information flow policy contains a partially ordered set of security levels $(\mathcal{L}, \leq)$ and a function $\mathrm{lbl} : Act \to \mathcal{L}$ assigning security levels to all inputs and outputs of the

$$\text{LOAD}\ \frac{R(l) \xrightarrow{i} P_l \qquad \text{if } \operatorname{lbl}(i) \leq l \text{ then } R'(l) = P_l \\ \text{else } R'(l) = R(l) \text{ for all } l}{(R, \emptyset) \xrightarrow{i} (R', \mathit{Upper}(i))}$$

$$\text{OUT-P}\ \frac{R(l) \xrightarrow{o} P \qquad \operatorname{lbl}(o) = l}{(R, l :: L) \xrightarrow{o} (R[l \mapsto P], l :: L)}$$

$$\text{OUT-C}\ \frac{R(l) \xrightarrow{o} C \qquad \operatorname{lbl}(o) = l}{(R, l :: L) \xrightarrow{o} (R[l \mapsto C], L)}$$

$$\text{DROP-P}\ \frac{R(l) \xrightarrow{o} P \qquad \operatorname{lbl}(o) \neq l}{(R, l :: L) \xrightarrow{\cdot} (R[l \mapsto P], l :: L)}$$

$$\text{DROP-C}\ \frac{R(l) \xrightarrow{o} C \qquad \operatorname{lbl}(o) \neq l}{(R, l :: L) \xrightarrow{\cdot} (R[l \mapsto C], L)}$$

Fig. 2: Semantics for secure multi-execution of a reactive system.

reactive system. The output $\cdot$ is invisible at all levels, and can be used to represent internal activity of the system. (For instance to return from a producer state to a consumer state without producing real output [10].)

A state of the wrapper is a tuple $(R, L)$, where

- $R$ is a function mapping security levels to states of the reactive system, $R : \mathcal{L} \to \mathit{State}$. $R(l)$ is the state of the sub-execution at level $l$.
- $L$ is the list of the levels of all the sub-executions that are in producer state (you can think of it as the scheduler's *ready queue*).

States $(R, \emptyset)$ are consumer states of the wrapper and states $(R, L)$ with $L \neq \emptyset$ are producer states. The initial state of the wrapper is a state $(R, \emptyset)$ such that for all $l \in \mathcal{L}$, the state $R(l)$ is the initial state of the original reactive system.

Fig. 2 shows the semantics of the wrapper. When a new input event $i$ arrives, it is passed to the copies at the levels in $\mathit{Upper}(i)$ (defined as a list of security levels higher or equal than the level of $i$), and the wrapper makes a transition to a producer state ([LOAD]). Once the wrapper is in producer state $(R, l :: L)$, it gives the sub-execution at level $l$ a chance to proceed. If this sub-execution produces an output at level $l$, the wrapper outputs it ([OUT-P] and [OUT-C]), otherwise a silent output $(\cdot)$ is produced instead ([DROP-P] and [DROP-C]). If the sub-execution at the level $l$ reaches a consumer state, then this level is removed from $L$ ([OUT-C] and [DROP-C]).

It is intuitively clear that this construction guarantees non-interference. Output at level $l$ is only produced from the sub-execution at level $l$, which only gets input at level $l$ or lower, so leaks from higher levels are impossible. On the other hand, the sub-execution at level $l$ receives identical input on level $l$ or lower. Therefore, if the program is non-interferent, then our wrapper still produces the same output as the original. It is possible that the order of outputs will be reordered though. We will discuss both of these aspects (*security* and *precision*).

### B. Security

First, we show formally that our technique guarantees non-interference: for any reactive system and any information flow policy, our wrapper will never produce information leaks.

Bohannon et al. proposed a bisimulation-based proof technique based on *ID-bisimulation* relation (written $\sim_l$)[10, Definition 4.1]. Our proof is based on their key theorem [10, Theorem 4.5] stating that if $Q \sim_l Q$ for all $l$, then $Q$ is ID-secure. In order to obtain ID-bisimulation relation on the wrapper states, we propose a definition of $l$-similarity.

*Definition 5.1:* The state of the wrapper $(R_1, L_1)$ is $l$-similar to the state $(R_2, L_2)$ (written $(R_1, L_1) \approx_l (R_2, L_2)$) iff a) $R_1 \approx_l R_2$ meaning $\forall l' \leq l : R_1(l') = R_2(l')$), and b) $L_1|_l = L_2|_l$, where $L|_l$ represents the list of levels $l'$ in $L$ such that $l' \leq l$.

Then we have proved the following key lemma.

*Lemma 5.1:* The $l$-similarity relation is an ID-bisimulation.

Since for every state $(R, L)$ of the wrapper we have $(R, L) \approx_l (R, L)$, we can finally use the Theorem 4.5 from [10] and prove the security theorem.

*Theorem 5.1 (Security):* All the states of the wrapper are ID-secure.

### C. Precision

On the other hand, we need to prove that our enforcement mechanism is precise: since it will sometimes modify the behaviour of programs, we need to prove that it does this in a sensible way, i.e. it does not observably modify behaviour for programs that already are secure. We show precise formal results to explain exactly what we mean by this.

First, we need to define what we mean by saying that our enforcement mechanism *does not observably modify the behaviour of programs*. It is important to notice that even for well-behaved programs, the wrapper can change the relative order of output events at different security levels. We assume that any observer will only observe at a single security level. This assumption is valid for the policies we will consider in Section VI. Then, we define the $\text{observer-indistinguishable}_l$ relation that relates input or output streams that "look the same" for observers at security level $l$. Like Bohannon et al., we use a coinductive definition to clearly specify this definition for infinite streams.

*Definition 5.2:* Define $\text{observer-indistinguishable}_l(S, S')$ coinductively with the following rules:

$$\text{observer-indistinguishable}_l([], [])$$

$$\frac{\operatorname{lbl}(s) \neq l \qquad \text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(s :: S, S')}$$

$$\frac{\operatorname{lbl}(s') \neq l \qquad \text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(S, s' :: S')}$$

$$\frac{\text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(s :: S, s :: S')}$$

This notion is weaker than Bohannon et al.'s ID-similarity. In fact, we have the following result:

*Lemma 5.2:* If $O \approx_l^{ID} O'$, then: $\text{observer-indistinguishable}_{l'}(O, O')$ for all $l' \leq l$.

Another notion we need is the projection of a finite stream at a certain security level $l$. The projection function $\pi_l$ removes from the stream those events that are at a level not below $l$.

*Definition 5.3:* Define, for finite $I_0$

$$\pi_l([]) = [] \quad \pi_l(i :: I_0) = \begin{cases} \pi_l(I_0) & \text{if } \mathrm{lbl}(i) \not\leq l \\ i :: \pi_l(I_0) & \text{if } \mathrm{lbl}(i) \leq l \end{cases}$$

Our enforcement mechanism produces observably equivalent outputs for those inputs for which the original reactive system is already "well-behaved" with respect to the security policy. We use the following precise definition:

*Definition 5.4:* Given a reactive system state $Q$ and a finite input $I$ and output $O$ such that $Q(I) \Rightarrow O$ we say that $Q$ *behaves securely for input* $I$ iff for all $l \in \mathcal{L}$, we have that $Q(\pi_l(I)) \Rightarrow O_l$ with $\text{observer-indistinguishable}_l(O, O_l)$.

These are the definitions we need to state the first of our precision theorems. The following theorem is the most detailed result, and shows that for those inputs for which the reactive system behaves securely, the corresponding wrapper produces results that are observationally equivalent.

*Theorem 5.2 (Precision for individual runs):* Suppose a given reactive system state $Q$ behaves securely for input $I$ and $Q(I) \Rightarrow O_Q$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all $l \in \mathcal{L}$, $L = \emptyset$ if $Q \in ConsumerState$ and $L = \mathcal{L}$ if $Q \in ProducerState$. For $O_W$ such that $W(I) \Rightarrow O_W$, we have that $O_Q \approx^{obs} O_W$.

This theorem is actually not a typical precision result for an information flow enforcement technique, because it does not require non-interference of the original system, as would be more typical (see e.g. Devriese and Piessens [9]). Instead, the theorem gives a sufficient condition for an individual execution to "behave securely" and produce observationally equivalent results. However, we can show that the previous theorem is stronger, by showing that if the original system was non-interferent, then all of its executions "behave securely".

*Lemma 5.3:* If a given reactive system state $Q$ is ID-secure, then it behaves securely for any input $I$.

This lemma easily leads to the following, more classical, precision theorem.

*Theorem 5.3 (Precision):* Suppose a given reactive system state $Q$ is ID-secure, and $Q(I) \Rightarrow O$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all $l \in \mathcal{L}$, $L = \emptyset$ if $Q \in ConsumerState$ and $L = \mathcal{L}$ if $Q \in ProducerState$. For $O'$ such that $W(I) \Rightarrow O'$, we have that $O \approx^{obs} O'$.

The stronger result is important in practice. Featherweight Firefox (without secure multi-execution) is never ID-secure: even if all scripts that have been loaded up to now behaved fine, somewhere in the future a malicious script might be loaded that leaks information. So the classical precision theorem does not apply, and it does not allow us to conclude precision for runs of the browser that actually behave well.

So what we need is a theorem that says: if the run of the browser up to some point behaved well, then our enforcement will not modify that run in an observable way. This is exactly what our first precision theorem does.

Note that we are only talking about precision here: security is never at stake. Featherweight Firefox with our enforcement mechanism will always be ID-secure. The point here is that we want to relate the behavior of the secured browser with the unsecured one, and this cannot be done with a classical precision theorem.

## VI. Information flow policies

We have implemented our information flow enforcement technique for Featherweight Firefox model in OCaml [1]. This implementation allows us to demonstrate valuable information flow policies for web browsers. The three basic policies we show demonstrate on the one hand the power of information flow policies, allowing us to define precisely the property that we want to enforce. On the other hand, our examples show that it is our technique that enforces the policies in a way that non-complying programs are dealt with as precisely as possible.

### A. Policy 1: High/Low Policy

A first, very simple but useful policy that can be enforced classifies all user inputs as H and all network outputs as L. This is essentially the policy we used in Section IV to explain our enforcement mechanism.

According to this simple High/Low policy, no public outputs are possible after secret inputs. It might seem that this will block any request to a website. But this is not the case. Intuitively, the reason why the request to "attacker.com" is being blocked is that it is made in response to a user input event, which is considered a private (H) information by our policy. Toward observers on the L security level, the policy enforcement therefore replaces this behavior by default behavior coming from the L execution, which is kept under the illusion that no user input has occurred.

Note that this policy is a very simple information flow policy, but already achieves something that previously was not possible. We can run a website making sure that certain user information is never leaked. For example, we can think of a "Keep all information in this field inside my browser" button that you can push to prevent information entered into a field from leaving your browser. The browser's policy enforcement could then use an enforcement technique like ours to guarantee security of the information, and in many cases without affecting the further behaviour of the site.

### B. Policy 2: Separating origins

The airplane tickets e-commerce site example is more typical for a general web site. In this scenario, a level of trust is assumed between the user and the company hosting the ticketing website, in order for the ticketing company to provide useful information or services. Nevertheless, the

[1] It can be accessed here: http://disi.unitn.it/~bielova/sme-firefox.

TABLE IV: Origin separation policy

| | | |
|---|---|---|
| User input | load_in_new_window(...)<br>input_text(user_window(dom), ...) | L<br>M(dom) |
| User output | window_opened<br>page_loaded(...)<br>page_updated(...) | H<br>H<br>H |
| Network input | receive(dom, ...) | M(dom) |
| Network output | send(dom, ...) | M(dom) |

TABLE V: Origin separation policy. M1 = M(“air.com”), M2 = M(“attacker.com”).

| | | | |
|---|---|---|---|
| L | | | load_in_new_window(”http://air.com”) |
| | L | H | ~~window_opened~~ |
| | | M1 | send(”air.com”, request_uri, cookies, ””)) |
| | H | H | window_opened |
| | | M1 | ~~send(”air.com”, request_uri, cookies, ””))~~ |
| M1 | | | receive(”air.com”, 0, cookie_updates, doc(age=0, ...)) |
| | M1 | H | ~~page_loaded(user_window, ”http://air.com”, doc(age=0, ...))~~ |
| | H | H | page_loaded(user_window, ”http://air.com”, doc(age=0, ...)) |
| M1 | | | input_text(user_window, 0, ”25”) |
| | M1 | H | ~~page_updated(user_window, doc(age=25, ...))~~ |
| | | H | ~~window_opened~~ |
| | | M1 | send(”air.com”, request_uri, cookies, ”?t=25”) |
| | | H | ~~window_opened~~ |
| | | M2 | ~~send(”attacker.com”, request_uri, cookies, ”?t=25”)~~ |
| | H | H | page_updated(user_window, doc(age=25, ...)) |
| | | H | window_opened |
| | | M1 | ~~send(”air.com”, request_uri, cookies, ”?t=25”)~~ |
| | | H | window_opened |
| | | M2 | ~~send(”attacker.com”, request_uri, cookies, ”?t=25”)~~ |

standard same-origin-policy (SOP) is not sufficient as it allows (in practice) this data to be sent anywhere.

We believe that the basic model of SOP is actually correct. When a user enters information on a website, it is typically his intent to disclose this information to the owner of that website, but not others. Likewise, information received from a website can be trusted to be sent back to this website but not to others.

A somewhat evident idea here is to use a security lattice with three types of levels: L, M(dom) for any domain *dom* and H. The L and H levels are smaller resp. bigger than all others and the M(...) domains are mutually incomparable. The M(dom) level is assigned to all network events originating from or going to this domain and to all user input events that contain information destined for a page on this domain. Output events going to the user are classified as H. This policy is summarized in Table IV.

Table V shows the execution of a prototypical airline ticketing website script under origin separation policy. A level of input event is in column 1, a level of sub-execution that receives this input is in column 2, and a level of an output is in column 3. We see that network output to “air.com” is now permitted to be influenced by information from user input.

Something interesting happens when we consider a page that tries to download a third-party script at page load time. Imagine that upon receiving an HTTP response receive (at level M1), the browser attempts to send the request for a third-party script (or image) at “remote.com”. Our policy marks input event receive as information that must be revealed only to the “air.com” domain. Hence the request to the “remote.com” should not be sent.

Of course, there is a good reason why the receive event should be classified at this level. If we suppose the page that is received represents the third step in the airline ticket purchasing process, and contains a summary of all data previously input by the user, then this is clearly information that we want to protect and the policy is correct to not allow this info to leak to third-party sites.

There is a tendency in information flow research to quickly turn to declassification [18] in such situations. Such techniques allow higher security information to be disclosed at lower levels under certain conditions. Declassification typically requires involvement from the sandboxed code and necessarily introduces extra complexity and weakens security guarantees. In our case, we are looking for a mechanism that can be transparently applied to existing code and declassification is not the best solution.

In fact, the problem in our example is that our information flow policy is not fine-grained enough. If we want to refine the SOP retaining maximum compatibility, we need to define a policy that does a better job of formalizing the assumptions in the current web security model. In this case, the policy does not capture the implicit notion that an HTML document contains information at different confidentiality levels. If the document specifies that it requires a certain script to function then this information must be permitted to leak to the website serving the script. In the next subsection, we discuss how this is possible without disclosing the entire document.

### C. Policy 3: Sub-input-event security policies

The key to solving the issue is to assign different labels to different parts of a single input event. One simple solution is to model such an input event as a number of separate input events, so that we can give each of these parts a different level. Then our enforcement mechanism and our security and precision theorems can be applied as before. An alternative, more intuitive way of thinking about this splitting of an input event (where different levels can see a different subset of the parts of the split event), is to consider security-level dependent projections that project an input event on the part of the event visible to a specific level. Space limitations keep us from discussing this policy in more detail. An extensive discussion can be found in the technical report [17]. With these final refinements, our approach realizes a substantial improvement over the standard SOP, while maintaining compatibility with typical cross-origin interactions in modern web applications.

## VII. Related Work

There is a large body of related work on information flow security in general, or on web security techniques in general.

We refer the reader to three good sources where these fields are surveyed. Sabelfeld and Myers [3] survey static techniques for information flow enforcement, and Le Guernic [4] surveys dynamic techniques. The PhD thesis of Martin Johns [11] gives a good survey of web security techniques and countermeasures for web-related vulnerabilites.

In some recent works, not yet covered in the surveys cited above, authors have developed dynamic [19] or static [8] techniques to enforce information flow security in a browser context. These techniques lack the precision guarantees of secure multi-execution, but on the other hand, secure multi-execution is likely to have a higher performance penalty.

In the rest of this section, we focus on the work that is most closely related to ours. A first very related line of work is the work by Bohannon et al. which has been discussed extensively in Section III. Next, there are several other security countermeasures that have strong similarities to our approach.

The technique of secure multi-execution was proposed by Devriese and Piessens [9] is the most closely related. This new technique is proved it to be sound and precise for a simple sequential programming language with synchronous I/O. Our paper extends their work to reactive systems and hence browsers. Interestingly, the formal guarantees we get are different. Whereas [9] can prove timing-sensitive non-interference, we have to settle for termination-insensitive non-interference. The main reason for this is that we are more restricted in the reordering of output events. On the other hand, we get a substantially stronger precision result. We show precision for any well-behaved run, whereas Devriese and Piessens can only prove precision for programs that are termination-sensitively non-interferent.

A similar approach was proposed by Capizzi et al. [15] where they run two executions of operating system processes for the H (secret) and L (public) security level. They limit themselves to this simple two-element poset, but they provide an actual implementation, and report on benchmarks.

In a very recent paper, Kashyap et al. [16], 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 used impacts the security properties offered.

Several very recent works have applied information flow analysis to web mashups. Magazinius et al. [5] propose an approach to construct a security lattice for mashups. Similarly to our work, where an element of the security lattice depends on the origin of the event, the authors of this paper defined the elements as sets of origins. The paper is focused on the definition of the policies, and does not focus on enforcement mechanisms. Li et al. also deal with mashups in their Mash-IF approach [6]. The security levels there consists of a tuple of sensitivity level and an origin. It is a practical approach, but no soundness or precision guarantees are provided.

## VIII. Conclusion and future work

This paper has studied the suitability of non-interference as a replacement for the same-origin-policy in browsers. We have shown that it is possible to enforce non-interference for a browser securely and precisely for a broad class of information flow policies. In addition we have shown that, even without any support for declassification, useful information flow policies for a browser can be defined.

In many cases, we can detect that the reactive system was not non-interferent to begin with, but it is future work to investigate what can be done in these cases. A clear possibility is to inform the user that this is the case, but we could also try to apply certain heuristics to improve precision.

An important remaining challenge is the development of efficient implementation techniques for our enforcement mechanism. It is also important to evaluate the impact of the proposed policies on real web sites: while the security benefits of a non-interference policy are high, there will be a price to pay. Even though we have shown by example that some level of compatibility with the current web can be maintained, it is to be expected that many detailed incompatibilities will show up, and evaluating the cost of these – and how they could be mitigated – is a key challenge for future work.

## References

[1] K. Singh, A. Moshchuk, H. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *SSP*, 2010.

[2] A. Bohannon and B. C. Pierce, "Featherweight Firefox: Formalizing the core of a web browser," in *WebApps*, 2010.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," in *JSAC*, vol. 21, 2003, pp. 5–19.

[4] G. L. Guernic, "Confidentiality enforcement using dynamic information flow analyses," Ph.D. dissertation, Kansas State University, 2007.

[5] J. Magazinius, A. Askarov, and A. Sabelfeld, "A lattice-based approach to mashup security," in *ASIACCS*, 2010.

[6] Z. Li, K. Zhang, and X. Wang, "Mash-IF : Practical Information-Flow Control within Client-side Mashups," in *DSN*, 2010, pp. 251–260.

[7] M. Louw, K. Ganesh, and V. Venkatakrishnan, "AdJail : Practical enforcement of confidentiality and integrity policies on web advertisements," in *USENIX*, 2010.

[8] R. Chugh, J. Meister, R. Jhala, and S. Lerner, "Staged information flow for Javascript," in *PLDI*, 2009.

[9] D. Devriese and F. Piessens, "Non-interference through secure multi-execution," in *SSP*, 2010.

[10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *CCS*, 2009.

[11] M. Johns, "Code injection vulnerabilities in web applications - exemplified at cross-site scripting," Ph.D. dissertation, University of Passau, 2009.

[12] ——, "On Javascript malware and related threats," *JCV*, vol. 4, pp. 161–178, 2008.

[13] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *NDSS*, 2007.

[14] B. Jacobs and J. Rutten, "A tutorial on (co)algebras and (co)induction," *EATCS Bulletin*, vol. 62, pp. 62–222, 1997.

[15] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla, "Preventing information leaks through shadow executions," in *ACSAC*, 2008.

[16] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and termination-sensitive secure information flow: Exploring a new approach," in *SSP*, 2011.

[17] N. Bielova, D. Devriese, F. Massacci, and F. Piessens, "Reactive non-interference for the browser: extended version," CS Dept., K.U.Leuven, Tech. Rep. CW602, February 2011. [Online]. Available: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW602.abs.html

[18] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *JCS*, vol. 17, pp. 517–548, October 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1662658.1662659

[19] T. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *PLAS*, 2010.