Reactive non-interference for the browser: a provable enforcement mechanism, sample policies and a proof of concept implementation

***Abstract*—Given a partially ordered set (po-set) of security levels, and a labeling of inputs and outputs with such levels, non-interference (or secure information flow) is a security property expressing that outputs of level l only depend on inputs that are labeled with a level smaller than l. In other words, there is no information flow from high (confidential) levels, to low (public) levels.**

For web browsers, as programs that interact intensely with a variety of principals, non-interference is an interesting security property, and several authors have studied how enforcement mechanisms for it can be incorporated in a browser, usually focusing on specific scenarios such as securing the flow of information towards advertisements, or securing mashups.

In this paper, we investigate the suitability of non-interference as a replacement for the baseline security policy of a browser, the *same-origin-policy*. We propose an enforcement mechanism that can enforce non-interference with respect to a broad class of security level posets for the full browser. We prove the security and precision of the enforcement mechanism, and implement it for the Featheweight Firefox browser model. Next, we investigate what security level posets are useful in a web context, and how inputs and outputs to the browser should be labeled. Somewhat surprisingly, our analysis shows that useful policies (that approximate but improve the current same-origin-policy) can be defined without any support for declassification.

***Keywords*-noninterference; web browser; dynamic enforcement**

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 can not interfere with each other in non-authorized ways. In today's browsers, this is achieved by enforcing the so-called *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 [14], it is also ambiguous and imprecise [1], and it fails to provide adequate protection for resources belonging to the user rather than to some origin [14]. 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 public outputs. In other words, secret inputs should not flow (directly or indirectly) to public outputs. Non-interference can be defined with respect to a more general *information flow policy*. Such a policy is a partially ordered set (po-set) of security levels l. The levels can be thought of as confidentiality levels: levels higher in the po-set will label more confidential information. All input channels and output channels of the program are labelled with such a security level, and the program is non-interferent if information only flows from inputs labelled l_i to outputs labelled 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 have been proposed. Sabelfeld and Myers [12] provide an extensive survey of static enforcement methods, and Le Guernic's PhD thesis [5] 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 [10], or to prevent private information to flow to advertisement providers [9].

Very recently, Bohannon et al. [2] proposed non-interference as a candidate replacement for the same-origin-policy. 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 provided a bisimulation-based proof technique to prove the soundness of enforcement mechanisms for reactive non-interference. In a later paper, Bohannon et al. [1] develop Featherweight Firefox, an extensive formalization of a web browser as a reactive system. They also provide an implementation of Featherweight Firefox.

A. Contributions of this paper

A first contribution of this paper is the development of an enforcement technique for reactive non-interference based on the technique of *secure multi-execution* [4]. We prove two major results:

- *Security* Featherweight Firefox (in fact any reactive system in the sense of Bohannon et al. [2]) is reactive non-interferent when executed under this secure multi-execution regime.
- *Precision* For inputs for which Featherweight Firefox is "well-behaved" with respect to the policy, execution under the secure multi-execution regime will not result in changes in observable behavior for an observer at any security level.

Further, we show the value of our technique for web browsers, by implementing it for Featherweight Firefox. To the best of our knowledge, our proposal is the first one to enforce a general non-interference policy for the browser as a whole.

Non-interference is parameterized by an information flow policy, so an interesting question is what policies are useful in a web browser (i.e. what should be the levels, and how should we assign them to inputs and outputs?). Interestingly, even without support for declassification, we show that many interesting and useful policies can be enforced for the browser and study their behavior on example scenario's. Some of the policies feature infinite security level posets, and we demonstrate how our implementation can support them. Other examples include useful policies that cannot be enforced by access control based systems, and policies that approximate the current same-origin-policy to maintain some form of compatibility with current browser policies. We argue that fine-grained policies are required to achieve more compatibility and discuss an extension of our technique to policies at a finer level than Featherweight Firefox input events.

In summary, the contributions of this paper are:

- The development of a provably sound and precise enforcement mechanism for reactive non-interference. Our precision results are stronger and more general than those in related work.
- The analysis of a variety of policies that can be enforced by this mechanism, thus providing evidence of the suitability of non-interference as a replacement for the current same-origin-policy.
- The implementation of this mechanism for the Featherweight Firefox browser model.

B. Plan of the paper

In the next section we specify the problem statement addressed in this paper. Then we summarize the results of Bohannon et al. which will be used in the rest of this paper in Section III. This section contains no original contributions, and can be skipped by readers familiar with Featherweight Firefox and reactive non-interference.

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. In Section VI, we discuss a variety of useful policies that can be enforced by our mechanism. We provide more details about our implementation for Featherweight Firefox in Section VII. Finally, we discuss related work, and conclude.

II. Problem statement

A browser interacts with a variety of web sites, and possibly executes Javascript code downloaded from some of these sites. Hence, a browser should enforce some security policy to make sure that these sites do not interfere with each other in undesirable ways. Today's browsers enforce the same-origin-policy, an access-control policy where browser resources are tagged with information about their origin, and access to resources is limited to code coming from the same origin.

The same-origin-policy has many problems, and has been criticized by many authors [7], [14]. Some of the issues, such as for instance the fact that different browser resources use different definitions of the notion 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 [14], 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 don't 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 information flow theoretic notions of non-interference would have advantages over the current access control policy.

A first, very simple, motivating example for an information flow policy 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 precalculating the amount of tax one has to pay in function of income, age, marital status and so forth. The service sends an HTML form for entering the user's information, and a JavaScript program that will calculate the tax due based on the information entered in the form.
The user would like assurance that the information he enters in the form does not leave his computer – not even to the website providing the service. Obviously, the same-origin-policy does not offer any protection for this scenario. 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 make sure that the script can not leak the private information to the network.

In many cases, the user will of course trust the website he is interacting with, and will be more concerned with information leaked to other sites.

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 assurance that this information does not leak to other sites.

The same-origin-policy provides some protection for this scenario: it will for instance make sure that scripts running in the user's browser and belonging to web pages from other origins can not access the information entered by the user. However, scripts that are part of the e-commerce web pages will have access and they can 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 [6]. 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 [11], 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 [9].

Another example scenario is a combination of the two examples above: some information or resources that the web application user provides are private to the user, others are intended to be shared only with the web application provider.

Example 3 (Flight ticket(revisited)): Even though the user trusts `http://www.air.com` with the information necessary to purchase a flight ticket, scripts from that site get access to other information that the user might want to protect, such as for instance geographical location (available to Javascript through the geolocation API), or the clipboard contents. Hence the user has partial trust in the site and shares some information, but wants assurance that (1) the information shared with the site will not leak to other sites (as in the previous example), and in addition (2) some user private information accessible to scripts remains private to the user (as in the first example).

An important additional challenge is the fact 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 very common to include content (most notably 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. It should for instance be possible for a script loaded from `b.com` into a page served from `a.com` to load images from any origin, since web advertising relies essentially on such scenarios. It should however be impossible for the script to leak information private to the user or to `a.com` in that scenario.

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 or related 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 sound and precise way (see the Related Work section for a detailed discussion). This paper proposes an enforcement mechanism, and proves it sound and precise.

Second, non-interference is parameterized with a policy: a partially ordered set of security levels, and an assignment of such levels to browser inputs and outputs. So an important problem is to select suitable policies. 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 cross-domain image and script loading.

III. Background

To address the first problem (the development of a general, sound and precise enforcement mechanism for a full browser), we need a formal model of a browser. In order to experiment with policies, this browser model should be executable. *Featherweight Firefox* is a browser model developed by Bohannon and Pierce [1] that satisfies these two requirements. It is a small-step operational semantics of a browser, that is implemented in OCaml[1]

In another paper, Bohannon et al. [2] have defined several variants of non-interference suitable for browsers, and proposed a bisimulation-based proof technique to establish one of these types of non-interference (called *ID-security* in their paper) . We will use their definition and proof technique to prove the soundness of our enforcement mechanism.

A. Reactive systems

At the highest level of abstraction, a browser is modeled as a *reactive system*, a particular kind of automaton that

[1]We used the version available from Aaron Bohannon's webpage: `http://www.cis.upenn.edu/~bohannon/browser-model/` where interested readers can find the full definition of the model.

reacts to input events by changing state and emitting output events.

Definition 3.1: A *reactive system* is a tuple

$$(ConsumerState, ProducerState, Input, Output, \rightarrow)$$

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

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

Consumer states are the states where the system is idle and waiting for inputs. A reactive system can only handle one input event at a time (thus correctly modeling the fact that Javascript event handlers are single threaded). Producer states are states where the system is processing, and from such producer states, the system can emit outputs. The definition allows for non-termination: it is possible that the system never returns to a consumer state.

Reactive systems transform streams of input events into streams of output events in the obvious way. A *stream* is defined as a coinductive interpretation of the grammar

$$S ::= [] \mid s :: S \tag{1}$$

where s ranges over stream elements. So a stream is a finite or infinite list of elements. 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 $Q(I) \Rightarrow O$ (Q transforms the input stream I to the output stream O) with the following rules:

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

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

B. Featherweight Firefox

The notion of reactive system is very abstract. To analyze potential security policies, we should define a browser model that concretizes the abstract states, inputs and outputs. The *Featherweight Firefox* browser model [1] does exactly that. It includes many browser features such as multiple browser windows; cookies; sending HTTP requests and receiving HTTP responses; essential HTML elements such as text boxes, buttons and links; building document node trees (i.e. a simple variant of the Document Object Model), and also the basic features of Javascript.

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. To understand our soundness proof further in the paper, it suffices to understand FF at the abstraction level of reactive systems. However, to understand the example policies and scenarios, some basic understanding of the full FF model is needed. We focus here on explaining the I/O events modeled by FF, the details of the internal browser state are less relevant.

When the browser is in a ConsumerState, it is ready to process any input event that could arrive. Input events can either come from the user (loading a URL in a new window, entering text in a text box, clicking a button), or from the network (receiving an HTTP response). Output events can also go to the user (web page is rendered or updated, window is closed) or to the network (sending HTTP request). The FF browser model defines precisely how the browser will react to each of these inputs by emitting outputs.

Some selected input and output events are shown in Table I. We simplify some syntactical constructs to give the reader a better understanding without unnecessary details.

Table I
SELECTED USER AND NETWORK I/O EVENTS.

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

An input event `load_in_new_window` models the case where a user navigates to some URL. When a user inputs some text into a text box this modeled by the event `input_text` . The second parameter is an index that uniquely identifies the text box that receives the input, and the third parameter is the text that was typed.

The display of a new window (`window_opened`) is an output event to the user and has no parameters since new windows are always created with a URL `about:blank`. Then after an html document is loaded or updated there may be visible changes of the rendered document on the screen. The `page_loaded` and `page_updated` events model these outputs to the user; the *rendered_doc* parameter of these events models a rendered document, and contains only elements that are visible to the user (for instance, script source code is not visible in a rendered html document).

Sending an HTTP request is modeled as an output event `send` of the browser. The request is sent to a particular *domain* and the tuple (*request_uri, cookies, string*) models a simplified version of an HTTP request.

The only input that can come from the network is the reception of an HTTP response. It contains a domain, an index that uniquely defines the open network connection on which the response arrives, updated cookies and the actual response content that consists of either an html document or a script.

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

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. investigate different notions of non-interference (for instance a termination sensitive and a termination insensitive notion), and use a notation that can distinguish the different notions. This paper only uses their notion of *ID-security*, a termination insensitive variant of non-interference. We specialize their definitions and notation to this case.

Let us assume that a po-set of security levels is given. The predicate $visible_l(s)$ models what observers of security level l can see: $visible_l(s)$ is true iff the stream element s is visible to an observer at level l. For instance, if we think of the input stream elements as arriving from input channels with a given security classification, then $visible_l(s)$ is true if l is higher than the classification of s's channel in the security ordering.

First, we define what it means for two (input or output) streams to be equivalent up to security level l[2].

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

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

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

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

Then, we can define when a reactive system (in a specific state Q) is secure.

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 $Q(I') \Rightarrow O'$.

As Bohannon et al. point out, this definition of security severely restricts the presence of non-determinism: for non-deterministic systems, a more intricate definition of non-interference will be necessary. Since FF is deterministic, we limit our attention in this paper to deterministic reactive systems, and the definition above satisfies our needs.

Definition 3.5: A reactive system is *deterministic* if

- for all $P \in ProducerState$ the following holds:

$$(P \xrightarrow{o} Q \wedge P \xrightarrow{o'} Q') \Rightarrow (o = o' \wedge Q = Q') \qquad (2)$$

- for all $C \in ConsumerState$ the following holds:

$$(C \xrightarrow{i} P \wedge C \xrightarrow{i} P') \Rightarrow P = P' \qquad (3)$$

The definitions in this section allow us to state our first goal for this paper more precisely: we want to construct an enforcement mechanism that ensures that FF is reactive non-interferent. Our non-interference proof will build on a result from Bohannon et al. [2]. They propose an interesting proof technique for establishing that a reactive system is non-interferent, based on the notion of $ID-bisimulation$.

Definition 3.6: An *ID-bisimulation* on a reactive system is a label-indexed family of binary relations on states (written $\sim_l$) with the following properties:

(a) if $Q \sim_l Q'$, then $Q' \sim_l Q$;
(b) if $C \sim_l C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i} P'$, then $P \sim_l P'$;
(c) if $C \sim_l C'$ and $\neg visible_l(i)$ and $C \xrightarrow{i} P$, then $P \sim_l C'$;
(d) if $P \sim_l C$ and $P \xrightarrow{o} Q$, then $\neg visible_l(o)$ and $Q \sim_l C$;
(e) if $P \sim_l P'$ then either
   - $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ implies $o = o'$ and $Q \sim_l Q'$, or else
   - $P \xrightarrow{o} Q$ implies $\neg visible_l(o)$ and $Q \sim_l P'$, or else
   - $P' \xrightarrow{o'} Q'$ implies $\neg visible_l(o')$ and $P \sim_l Q'$.

They show that the existence of an ID-bisimulation entails non-interference.

Theorem 3.1 ([2]): If $Q \sim_l Q$ for all l, then Q is ID-secure.

This theorem will be a key building block of our soundness proof: we will establish non-interference for our enforcement mechanism by showing the existence of an ID-bisimulation.

IV. Informal overview

The enforcement mechanism we propose in this paper is based on a relatively new dynamic technique for achieving non-interference: secure multi-execution [3], [4]. The core idea of this mechanism is to execute the program multiple times (one copy of the program for each security level), and to make sure that (1) outputs of a given level l are only done in the execution at level l (outputs are suppressed in other copies), and (2) inputs at a level l are only done at level l (for the other copies above l, the values that were input by level l are reused, whereas copies that are not above l are fed a default input value). Hence the copy that does output at level l only sees inputs of levels below l and hence the output could not have been influenced by inputs of a higher level. Non-interference follows easily from this observation.

[2]We take an original definition of VS-similarity and put it in the definition of ID-similarity since it was proven that $S \approx_l^{ID} S'$ iff $S \approx_l^{VS} S'$ [2].

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

Figure 1. Javascript code example

Devriese and Piessens [4] have worked out this enforcement mechanism for the case of a simple sequential programming language with synchronous I/O, and have proven security and precision in that setting. Capizzi et al. [3] have implemented the technique at the level of operating system processes for the case of two security levels H (secret) and L (public).

The mechanism we propose adapts this technique to reactive systems, and we prove its security (somewhat weaker than what Devriese and Piessens have shown in their setting, because we lose termination- and timing-sensitivity), as well as its precision (somewhat stronger than the result by Devriese and Piessens, because we show precision under 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 Figure 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 textfield c. We can assume this Javascript code is part of an event handler that fires whenever the user changes the contents of a or b.

The code in the figure also shows a potential attack: the script will leak the (secret) sum to `http://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 from Section II 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.

Table II shows the behavior of Featherweight Firefox on this example. Since Featherweight Firefox does not support images, we simulate the information leak through a page load instead of an image load, which is from the point of view of information flow security the same thing.

If we assume that the inputs to the textfields have a high security level (H), and the output to `http://attacker.com` has a low security level (L), then this program is clearly not secure: high inputs leak to low outputs. How will our enforcement mechanism close this leak?

First, we have to assign security levels to all inputs and outputs of Featherweight Firefox. For this example, we assign H to `input_text` events and L to all other events. Next, following the idea of secure multi-execution, we run several copies of the web browser, one for each security level. Input events of level l are only processed by the copies with a level above or equal to l. Output events of level l are only produced in the copy at level l. Tables III and IV show what happens in both copies. The tables also show what inputs and outputs get suppressed in each level. For instance, for the low copy, the following things get suppressed: (1) the input events of level H (and hence also all output events that would have been the result of that input event), and (2) the output events at level H.

Table III
RUN OF L COPY OF THE BROWSER.

	Input/Output
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*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_inline))~~
H	~~`input_text`(*user_window*, 1, "2")~~
H	↳ ~~`page_updated`(*user_window*, *doc*(a=0, b=2, c=2, js_inline))~~
H	↳ ~~`window_opened`~~
L	↳ ~~`send`("attacker.com", *request_uri, cookies*, "?t=2")~~

Table IV
RUN OF H COPY OF THE BROWSER.

	Input/Output
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*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_inline))
H	`input_text`(*user_window*, 1, "2")
H	↳ `page_updated`(*user_window*, *doc*(a=0, b=2, c=2, js_inline))
H	↳ `window_opened`
L	↳ ~~`send`("attacker.com", *request_uri, cookies*, "?t=2")~~

The offending output to `http://attacker.com` is suppressed, as the L copy never gets the input event where the user is typing secret data in the text box. Even if the script would try to send the contents of a and b later in the execution in response to a L input, the actual output sent to "`attacker.com`" would only contain the sum of the default values in both textfields. There is never any information flow from H inputs to L outputs.

V. FORMALIZATION

We propose to apply an approach of secure multi-execution to a reactive system first. Given an information-

Table II
CORRESPONDENCE BETWEEN USER'S ACTIONS AND I/O OF FEATHERWEIGHT FIREFOX

Description of user actions and network events	Input/Output of Featherweight Firefox
User opens a url of the tax calculator in a new window, as a result the new window is opened and an HTTP request is sent	`load_in_new_window`("http://taxcalc.com") ↪`window_opened` ↪`send`("taxcalc.com", *request_uri, cookies*, ""))
Network sends an HTTP response with html doc containing fields *a*, *b*, *c* and inlined javascript function	`receive`("taxcalc.com", 0, *cookie_updates*, *doc*(a=0, b=0, c=0, js_inline)) ↪`page_loaded`(*user_window*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_inline))
User types "2" into a text box *b*. This triggers the Javascript event handler to execute the attack	`input_text`(*user_window*, 1, "2") ↪`page_updated`(*user_window*, *doc*(a=0, b=2, c=2, js_inline)) ↪`window_opened` ↪`send`("attacker.com", *request_uri, cookies*, "?t=2")

flow policy, we build a new reactive system that we call a *wrapper*. As in previous work [4], the wrapper internally runs multiple copies (*sub-executions*) of the original reactive system: one for each security level. When the wrapper consumes an input event, its security level is determined, and the input event is passed to those sub-executions that are allowed to see it, i.e. the sub-executions at a level higher than the input event's level. When a sub-executions at a security level produces an output event, its security level is determined and only if the two levels are the same, the output event is produced by the wrapper.

Because of space constraints, we do not provide proofs of our theorems. Full proofs are available in a separate technical report.[3]

A. Secure multi-execution of reactive systems

We assume that the information-flow policy is given. It contains a partially ordered set of security levels $\mathcal{L}, \leq$. Also the policy defines a function $\mathrm{lbl} : Act \to \mathcal{L}$ that assigns security levels to all inputs and outputs of the reactive system. This function lbl corresponds to Bohannon et al.'s predicate $visible_l$ such that $visible_l(s)$ iff $\mathrm{lbl}(s) \leq l$ and $s \neq \cdot$. The output $\cdot$ is an output invisible to 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.)

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

- R is a function mapping security levels to states of the reactive system, $R : \mathcal{L} \to State$. $R(l)$ 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.

We define the semantics of the wrapper in Figure 2. When a new input event i arrives, it is passed to the copies at the levels in $Upper(i)$ (defined as the list of security levels higher than the level of a given input i), and the wrapper makes a transition to a producer state (rule [LOAD]). Once the wrapper is in producer state (R, L) it takes the first security level l from the list of levels L and gives the copy at this level a chance to proceed. If it produces an output at level l, it is also produced by the wrapper (rules [OUT-P] and [OUT-C]), otherwise a silent output $(\cdot)$ is produced instead (rules [DROP-P] and [DROP-C]). If the copy at the selected level l goes to a consumer state, then this level is removed from the L (rules [OUT-C] and [DROP-C]).

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

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

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

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

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

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

It is intuitively almost trivial to see why this construction guarantees non-interference. Output at any level l is only produced from the sub-execution at level l, which only gets to see input at level l or lower, so any leaks from input at higher levels is impossible. On the other hand, the sub-execution at a level l receives identical input on level l or lower as the original. Therefore, if the program is such

[3]Note to reviewers: because of anonymization, we do not cite the report here, but we can make it available to the PC chairs on request.

that higher-level input does not influence lower-level output, then our construction will still produce 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 (*soundness* and *precision*) in the next subsections.

B. Soundness

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

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

With Bohannon et al.'s bisimulation-based proof technique [2, Theorem 4.5], it suffices to prove that there exists an ID-bisimulation $\approx_l$ such that for every state of the wrapper (R, L), we have that $(R, L) \approx_l (R, L)$. We use the following bisimilarity relation $\approx_l$. For a list of security levels L, we use the notation $L|_l$ to represent the list of levels l' in L such that $l' \leq l$.

Definition 5.1: The state (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

- $R_1 \approx_l R_2$ meaning for all $l' \leq l.\, R_1(l') = R_2(l')$), and
- $L_1|_l = L_2|_l$.

To prove that this is a bisimulation, we basically need to show that l-similar states produce outputs that are equal at level l or lower, and that the relation is maintained when they receive inputs that are equal up to level l.

Lemma 5.1: This l-similarity relation is an ID-bisimulation.

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 respect the policy. We will show precise formal results to explain exactly what we mean by this.

First, we need to define what we mean when saying that our enforcement mechanism *does not observably modify the behaviour of programs*. Important to notice is that even for well-behaving programs, the wrapper can change the relative order of output events at different security levels. With the assumption that any observer will only ever observe at a single security level, 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:

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

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

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

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

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 $l' \leq l$.

The notion of similarity that we will use for our precision results requires that the wrapper's output "looks the same" as the original output for observers at any one level. We define $S \approx_l^{obs} S'$ if $\text{observer-indistinguishable}_{l'}(S, S')$ for all $l' \leq l$. Note how these notions allow for changes in the relative order of events on different security levels.

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

Definition 5.3: Define, for finite I_0

$$\pi_l([]) = []$$

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

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 Q and a finite input I and output O such that $Q(I) = O$, we say that Q *behaves securely for input* I iff for all $l \in SecurityLevel$, we have that $Q(\pi_l(I)) = O_l$ 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 Q behaves securely for input I and $Q(I) = O_Q$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all l, $L = \emptyset$ if $Q \in ConsumerState$ and $L = \mathcal{L}$ if $Q \in ProducerState$. For $O_W = W(I)$, 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 [4]). 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 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 Q is ID-secure, and $Q(I) = O$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all l, $L = \emptyset$ if $Q \in ConsumerState$ and $L = \mathcal{L}$ if $Q \in ProducerState$. For $O' = W(I)$, we have that $O \approx^{obs} O'$ for any security level l.

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 that we cannot do that with a classical precision theorem.

VI. Information flow policies

The implementation of our information flow enforcement technique for Featherweight Firefox allows us to demonstrate some different information flow policies that are valuable as browser security policies. We think that the three basic policies we show demonstrate on the one hand the power of information flow policies, allowing us to define precisely (contrary to traditional access control policies) the property that we want to enforce. On the other hand, our examples also show that it is our enforcement technique that enforces the policies in such a way that non-complying programs are dealt with as precisely as possible (not just terminating them like traditional information flow policy enforcement techniques would).

The importance of current web applications calls for strong guarantees, but the enormous amount of legacy software out there calls for a highly compatible solution. We think that it is the combination of the accuracy of information flow policies together with the precise support for non-complying programs that makes a nice fit for the requirements of a browser security policy context.

Table V
Simple High/Low policy

User input	`load_in_new_window`(*url*)	L
	`input_text`(*user_window, nat, string*)	H
User output	`window_opened`	H
	`page_loaded`(*user_window, url, rendered_doc*)	H
	`page_updated`(*user_window, rendered_doc*)	H
Network input	`receive`(*domain, nat, cookie_updates, resp_body*)	L
Network output	`send`(*domain, request_uri, cookies, string*)	L

A. High/Low Policy

Let us take another look at the tax calculator example. In this case, we would like the browser to guarantee to the user that his input does not leave his computer. However, it is our goal to do this without breaking the legacy website that uses existing DOM APIs and might interact with internet servers for downloading scripts and images.

Already for this simple scenario, an access control policy is not fine-grained enough to achieve these goals. An access control policy can prevent information leaks through server requests only by allowing requests to pass or not based on their content. Unlike an information flow policy, it cannot reason about what input the requests' content was constructed from and as such, it cannot distinguish requests that actually leak information from those that don't.

An information flow policy can be more fine-grained. In this case, the policy could classify all user input as secret information (H) and all network requests as public (L), disallowing URLs for dynamically added images to depend on secret information. Table V shows the classification of some I/O events. The policy then states that output events at level L (public) are not allowed to be influenced by input events at level H (secret).

Figure 3 shows a table representing the execution of a prototypical script for the tax calculator example using our enforcement technique for the simple High/Low policy from Table V. We see the browser responding in the normal way to the user navigation and the load of the script page. In response to the user entering text in one of the text boxes, the script modifies the page and the browser shows the modified page to the user. However, the script is malicious and tries to leak the user's information by triggering a navigation to a page under the "`attacker.com`" domain.

However, our enforcement mechanism modifies this behaviour: we see that depending on the level of each input event, it is either passed to both or only one of the sub-executions at levels L and H. Each sub-execution produces

Figure 3. Executing a tax calculator script containing a malicious leak, enforcing the simple High/Low policy from Table V.

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

output events, but only the output events at its own level are actually produced by the wrapped system. This effectively prevents the information leak that the script is trying to trigger, as we can see in the example: the request to "attacker.com" is not produced, since the low execution does not receive the input_text input event and the high execution will not trigger this output. To the people behind "attacker.com", it seems as though the user loads the page, but never inputs any text.

Even though the behaviour towards the external website is being modified, the user in this case sees exactly the same behaviour as he would without the security mechanism. As long as future user-observable behaviour does not depend on the reactions of untrusted observers to information leaking requests, this is likely to be okay. For example in the case that the leak was the result of an XSS-attack [11], this is likely to be the case.

In this example, it might seem that our simple High/Low policy will block any request to a website. However, this is not the case. Intuitively, the reason that the request to "attacker.com" is being blocked is that it is being made in response to a user input event, and the fact that the user has performed a text input is defined to be private 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 giving the illusion that no user input has occurred.

We can illustrate this observation by demonstrating another script that makes legitimate use of external requests. Let's suppose that the tax calculator script needs information from a third-party website (for example, an up to date table of tax rates for different income ranges). In table VI, we show what happens if the script requests to download such a table from a server at "remote.com" after the page has loaded.

In this case, we see that the legitimate request to "remote.com" is allowed to pass without any problem.

Table VI
SIMPLE HIGH/LOW POLICY, THIRD-PARTY SCRIPT

	Input/Output		
L	load_in_new_window("http://taxcalc.com")		
	L	H	~~window_opened~~
		L	send("taxcalc.com", *request_uri, cookies*, ""))
	H	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_remote))		
	L	H	~~page_loaded(*user_window*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_remote))~~
		L	send("remote.com", *request_uri, cookies*, ""))
	H	H	page_loaded(*user_window*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_remote))
		L	~~send("remote.com", *request_uri, cookies*, ""))~~
L	receive("remote.com", 0, *cookie_updates*, js_function)		
	L		
	H		
H	input_text(*user_window*, 1, "2")		
	H	H	page_updated(*user_window*, *doc*(a=0, b=2, c=2, js_remote))
		H	window_opened
		L	~~send("attacker.com", *request_uri, cookies*, "?t=2")~~

Table VII
SIMPLE HIGH/LOW POLICY, STEALING COOKIES

	Input/Output		
...			
L	receive("taxcalc.com", 0, ("lang", "ru"), *doc*(a=0, b=0, c=0, js_remote))		
	L	H	~~page_loaded(*user_window*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_remote))~~
		L	send("remote.com", *request_uri, ()*, ""))
		L	send("attacker.com", *request_uri, ()*, "?lang=ru"))
	H	H	page_loaded(*user_window*, "http://taxcalc.com", *doc*(a=0, b=0, c=0, js_remote))
		L	~~send("remote.com", *request_uri, ()*, ""))~~
		L	~~send("attacker.com", *request_uri, ()*, "?lang=ru"))~~
...			

When its response comes in, both the L and H execution see this and the program behaves as intended. This shows that indeed, we get what we ask for: our policy specifies only that user input mustn't influence network output, and indeed we see that network output that is not influenced by user input behaves as intended.

Finally, the simple High/Low policy is designed specifically for a case like a tax calculator website where all interaction with the user can be assumed to occur inside the browser. Specifically, the policy assumes all information coming from the network as public data. It is important to remember, when applying this policy, that this policy will not provide any protection for information that is classified as public, like for example page content or cookies. As an example, table VII shows a case where a cookie tracking the user's language is leaked. This is not a limitation of our enforcement mechanism, but a limitation of the specific policy that it is enforcing here.

Note that this policy is a very simple information flow policy, but already achieves something that was previously

Table VIII
ORIGIN SEPARATION POLICY

User input	`load_in_new_window`(*url*) `input_text`(*user_window, nat, string*)	L M(...)
User output	`window_opened` `page_loaded`(*user_window, url, rendered_doc*) `page_updated`(*user_window, rendered_doc*)	H H H
Network input	`receive`(*dom, nat, cookie_updates, resp_body*)	M(dom)
Network output	`send`(*dom, request_uri, cookies, string*)	M(dom)

not possible with simple access control policies. We can run a website making sure that certain user information is never leaked. It is not hard to think of interesting extensions of this idea for real-life browser scenario's. We imagine for example 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 make sure that the rest of the behaviour of the site is hopefully unmodified.

B. 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. Nevertheless, the 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 the SOP is actually correct. When a typical 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. We think that using information flow enforcement techniques such as the one described in this paper, a replacement for the SOP can be defined that does achieve the intended information protection, with sufficient backwards compatibility for much of the code currently "out there".

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(domain) 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 VIII.

Table IX shows the execution of a prototypical airline ticketing website script under the origin separation policy from table VIII. We see that network output to "`air.com`" is now permitted to be influenced by information from user input in the corresponding web page.

Table IX
ORIGIN SEPARATION POLICY. M1 = M("AIR.COM"), M2 = M("ATTACKER.COM").

	Input/Output		
L	`load_in_new_window`("http://air.com")		
	L	H	~~`window_opened`~~
	L	M1	`send`("air.com", *request_uri, cookies*, ""))
	H	H	`window_opened`
	H	M1	~~`send`("air.com", *request_uri, cookies*, "")~~)
M1	`receive`("air.com", 0, *cookie_updates*, *doc*(age=0, ..., js_inline))		
	M1	H	~~`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_inline))~~
	H	H	`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_inline))
M1	`input_text`(*user_window*, 0, "25")		
	M1	H	~~`page_updated`(*user_window*, *doc*(age=25, ..., js_inline))~~
		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, ..., js_inline))
		H	`window_opened`
		M1	~~`send`("air.com", *request_uri, cookies*, "?t=25")~~
		H	`window_opened`
		M2	~~`send`("attacker.com", *request_uri, cookies*, "?t=25")~~

Table X
ORIGIN SEPARATION POLICY, THIRD-PARTY SCRIPT. M1=M("AIR.COM"), M2=M("REMOTE.COM").

	Input/Output		
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, ..., js_remote))		
	M1	H	~~`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_remote))~~
		M2	~~`send`("remote.com", *request_uri, cookies*, "")~~)
	H	H	`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_remote))
		M2	~~`send`("remote.com", *request_uri, cookies*, "")~~)
...			

Something interesting happens when we consider a page that tries to download a third-party script at page load time. In Table X, we see that our security mechanism prevents the request for the third party script from being sent at all, very likely breaking the site's behaviour.

Let us see why this happens. The request for the third-party script on host "`remote.com`" is produced in response to the `receive` input event representing the receival of the website document, since that is where it is specified that the third-party script should be loaded. However, our policy marks this input event as information that must only be revealed through user output and network output to the

"air.com" domain. So our security enforcement cannot be blamed for breaking the website, since it is only executing what the policy specified. So that must mean that the policy is wrong and we should have classified the receive input event on a different security level?

Unfortunately, there is a very good reason why it 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 just allow this info to leak to third-party sites.

One solution would be to provide support for declassification, and to declassify parts of the page. However, declassification is complex [**?**] and makes it hard to understand the policy that is still being enforced. So we prefer to avoid it.

A possible answer is that the information flow policy is not fine-grained enough. If we want to refine the SOP retaining maximum compatibility, then we need to define a policy that does a better job of formalizing the assumptions in the current web security model. In this case, there is 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 in question. However, we have to make sure that the rest of the document cannot be leaked in the process. The next subsection discusses how to incorporate this in our mechanism.

C. Sub-input-event security policies

The key to solving the issue is to be able 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 splitted event), is to consider security-level dependent projections that project an input event on the part of the event visible to a specific level. We discuss our solution from this angle.

Whereas the policies we discussed in previous subsections have been implemented for Featherweight Firefox, implementing the solution we discuss in this section is ongoing work.

So far, the information flow policy was defined by the function $lbl : Act \to SecLevel$ returning the security level for any input or output event.

Alternatively, we can give a more flexible definition of a security policy by restricting the lbl function to output events and additionally requiring a set of projection functions $\{\pi_l : Input \to Input \cup \{Suppress\} \mid l \in SecLevel\}$.

These projection functions have to be idempotent and such that the projection function π_H at maximum level H is the identity function and for all $l \leq l'$ and input events i and i', we have that $\pi_l(i) = \pi_l(i')$ whenever $\pi_{l'}(i) = \pi_{l'}(i')$.

The projection function π_l intuitively defines the view of an input event at security level l (Suppress is a keyword signaling that the input event should not be visible at this level at all) and what we've been doing up to now actually corresponds to projection functions defined as follows:

$$\pi_l(i) = \begin{cases} i & \text{if } l \geq lbl(i) \\ Suppress & \text{if } l \not\geq lbl(i) \end{cases}$$

Based on these projection functions, we can adapt the formal definitions of non-interference and our enforcement technique in an obvious way. The wrapper constructed by our enforcement technique will no longer send an input event i to all sub-executions at levels $l \geq lbl(i)$, but will instead send the projection $\pi_l(i)$ of the input event to all sub-executions at levels l such that $\pi_l(i) \neq Suppress$. We conjecture that analogues for all of our results hold for these definitions.

D. Security level per element

With these sub-input-event policies, we can refine our solution for the origin separation policy to make it align more closely with the assumptions that are implicit in the web model. In particular, for an input event $i = \texttt{receive}(dom, nat, cookie\_updates, body)$, we propose to define the projection functions as follows:

$$\pi_l(i) = \begin{cases} i & \text{if } l = H \text{ or } l = M(dom), \\ \texttt{receive}(dom, nat, \{\}, proj_{dom'}(body)) & \\ & \text{if } l = M(dom'), dom \neq dom', \\ Suppress & \text{if } l = L, \end{cases}$$

where $proj_{dom}(body)$ projects a HTML document onto an almost empty document with only public information remaining. It is important that script tags referencing scripts on domain dom' are also kept. The assumptions here is that when a server generates a page for a user, most information is intended to be kept private, where private means that it can only flow to the user and back to the originating server. We make an exception only for that information that the server must have intended to be leaked to certain destinations. For example, when the server links to a script on a third-party domain, then the server is well aware that this will trigger a request to the domain in question. Therefore, there is no harm in disclosing this fact on the security level corresponding to that third-party domain.

If we apply this relaxed policy to the "air.com" website script with a third-party script, then contrary to Table X, the request to "remote.com" will be sent properly, in the M2 execution. A question is then what should happen when

Table XI
FINE-GRAINED ORIGIN SEPARATION POLICY, THIRD-PARTY SCRIPT.
M1=M("AIR.COM"), M2=M("REMOTE.COM").

	Input/Output		
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, ..., js_remote))		
	M1	H	~~`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_remote))~~
		M2	~~`send`("remote.com", *request_uri, cookies*, ""))~~
	M2	H	~~`page_loaded`(*user_window*, "http://air.com", $proj_{M2}$(doc(age=0, ..., js_remote)))~~
		M2	`send`("remote.com", *request_uri, cookies*, "")
	H	H	`page_loaded`(*user_window*, "http://air.com", *doc*(age=0, ..., js_remote))
		M2	~~`send`("remote.com", *request_uri, cookies*, ""))~~
L	`receive`("remote.com", 0, *cookie_updates*, js_function)		
	...		

the response for this event comes in. If we classify it at security level M("remote.com"), then we are saying that its content must not leak to the "`air.com`" domain, which is a strong restriction. If the third-party script is a javascript library like JQuery, then it is clear that this restriction is too strong and will break the functionality of the website. Hence the straightforward solution is to classify scripts loaded from script tags in the document on the L security level (loosely corresponding to the current browser policy).

In some cases, it might be desirable to give these scripts a higher classification, for instance if the third-party script is the JSON result of a web service call. In that case, protection of the returned script is valuable, since there might be a form of CSRF "Javascript hijacking" [] attack taking place.

Hence, the classification of this response is a trade-off between security and compatibility. This is a result of the ambiguities in the current web security model.

Taking inspiration from existing heuristics to protect client-side against CSRF attacks [], we believe an interesting path for future work is to investigate if a heuristic labeling algorithm can be defined that will classify the contents of loaded third-party scripts in a reasonably secure, yet compatible way, using information like the security properties of the protocol it was downloaded over (e.g. HTTPS vs. HTTP), whether cookies were sent along with the request, etc. If cookies are available to be sent along with the request, it might even be a good idea to request the script twice (resp. with and without cookies) and expose the two results on different levels. Experiments with real websites are needed to flesh this out further.

With a fine-grained policy as described here, the prototypical airline ticketing website using a remote script would behave as in Table XI. The remote script is now working fine, yet user input and document data is protected suitably.

VII. IMPLEMENTATION

We have implemented our enforcement mechanism based on the idea of secure multi-execution for the Featherweight Firefox browser model in OCaml and tested it on our policies[4].

The operational semantics defined in Fig. 2 is implemented at the level of a reactive system. Even though the construction is relatively simple, we had to address several interesting issues.

One problem is the fact that policies can potentially have an infinite number of levels, for instance in the policies where there is one level per origin. We have solved this by adding additional levels lazily. The following lemma shows that this is sound (we prove it for finite but unbounded streams).

Recall that the state of the wrapped system is a tuple (R, L), where R is a pointer to the running copies of the browser and $L \neq \emptyset$ is a list of levels at which the browser copies are in producer state. When waiting for a new input, the state of the wrapped system is $(R, \emptyset)$ because all the current states of the browser copies are in consumer state.

Lemma 7.1: Suppose the current state of the wrapped system is $W = (R, \emptyset)$. If R is defined for l and l' and the input I is such that $\pi_l(I) = \pi_{l'}(I)$ (the input I looks the same at l and l'), then $R(l) = R(l')$.

Hence, when the browser first sees an input of level l (e.g. the user visits a new website), we can lazily add the sub-execution at that level based on this lemma. We take the existing sub-execution at level l' such that $\pi_l(I) = \pi_{l'}(I)$ (where I is the input arrived so far) and use a copy of it for l. In the implementation it would usually mean that l' is one level lower than l in the security lattice.

In particular, we implemented this idea for the policy with security levels of three types: L, M(dom) and H. In the beginning of the run the wrapped system has only two copies of the original browser: at levels H and L. If the input at a new level l arrives, then in the security lattice it is such that L $< l <$ H. This means that all the inputs I arrived so far look the same at new level l and the lowest level L: $\pi_l(I) = \pi_L(I)$. Hence, we take the Lth copy of the browser and use it for a copy at new level l.

Do copies at level l' always exists for an input at a new level l such that $\pi_l(I) = \pi_{l'}(I)$? Unfortunately not. If the new input level l is such that there exist two incomparable levels $l_1 < l$ and $l_2 < l$, then $\pi_l(I) \neq \pi_{l_1}(I)$, $\pi_l(I) \neq \pi_{l_2}(I)$. In this case the lth copy of the browser should be a combination of the l_1th and l_2th copies. Note that this cannot happen if we consider security level lattices (instead of general posets) and if we make sure that the set of levels for which we run a copy remains a sub-lattice after adding a new level.

[4]Concretely an information-flow policy is implemented as an OCaml function mapping input and output events to security levels.

Our implementation only considers the security po-sets of the forms we discussed in the previous sections, and for these po-sets, we can always lazily add new levels. This is because the security level posets we consider are in fact lattices and any set containing H and L is a sub-lattice.

Similarly, it can happen that the output event o produced by the lth copy of the browser has a level l' and is not in the list of levels at which the wrapped system is currently running the browser copies. This means that there is no copy of the browser at level l' that will be able to output o. Hence, if the level l' of o is such that $l < l'$ then we allow the lth copy to output o because this does not violate the noninterference (higher inputs should not influence lower outputs). However, we do not add a new copy of the browser at level l' to be run by the wrapped system because it is not necessary for the correctness of our approach.

We have implemented the solutions discussed in this section and provided an implementation for the High/Low policy as well as for the separating origins policy described in Section VI. As we already argued, more sophisticated policies such as "sub-input-event" are ongoing work as they need a substantial intervention in the model of the Featherweight Firefox.

VIII. 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 [12]) survey static techniques for information flow enforcement, and Le Guernic [5] surveys dynamic techniques. The PhD thesis of Martin Johns [7] gives a good survey of web security techniques and countermeasures for web-related vulnerabilites.

In the rest of this section, we focus on the work that is most closely related to ours.

A first, a very related line of work is the work by Bohannon et al. that 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 proposed by Devriese and Piessens [4] is the most closely related. These authors proposed secure multi-execution for enforcing noninterference, and proved it to be sound and precise. They show these results for a simple sequential programming language with synchronous I/O. Our work extends theirs, by showing how the same technique can be applied to reactive systems and hence browsers. Interestingly, the formal guarantees we get are different. Whereas Devriese and Piessens 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. [3] 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 po-set, but they provide an actual implementation, and report on benchmarks.

Some other web-browser security techniques solve different problems, but use techniques that look like ours.

One recently proposed technique called AdJail [9] is particularly aimed at the information flow between the user data displayed on the web page and the third-party advertisements. Similarly to the secure multi-execution and to the shadow execution approaches, the authors propose to have a shadow copy of the web page where all the interactions between the ad script and the original page are controlled. They also report on an implementation. This paper is an excellent example of how shadow executions can be used to address specific web security issues at a reasonable performance cost, but obviously the scope of the protection offered is smaller than for our proposed countermeasure.

Doppelganger [13] is another example of a similar approach with a very specific focus. It focusses on keeping control over HTTP cookies. The approach suggests to have two copies of the running web page: with and without cookies. It defines the difference between the pages in such a way that in certain case the enforcement mechanism can decide automatically whether keeping cookies is important for the correct functionality of the web page. This technique, however, is concerned only about the HTTP cookies and does not cover general information flow or other browser functionalities.

Several very recent works have applied information flow analysis to web mashups. Magazinius et al. [10] propose an approach to construct a security lattice for mashups. Similarly to our approach, where an element of security lattice depends on the origin of the event, the authors of this paper defined the elements as sets of origins. So the security lattice consists of elements with one origin (for events) and elements with all possible combinations of the origins. In cases where different origin domains have to communicate, the approach relies on declassification. The paper is focused on the definition of the policies, and does not focus on enforcement mechanisms.

Li, Zhang and Wang also deal with mashups in their Mash-IF approach [8]. 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.

IX. 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 (even including policies with an infinite number of levels). In addition we have shown that, even without any support for declassification, useful information flow policies for a browser can be defined.

An important remaining challenge is the development of efficient implementation techniques for our enforcement mechanism (or alternative secure and precise mechanisms). Another important item for future work is the evaluation of the impact of the policies we proposed on real web sites: while the security benefits of a non-interference policy are high, it is to be expected that 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.

Still, we do believe that the results reported in this paper provide evidence that it is worthwhile to go further down this road, and do the substantial effort of integrating non-interference mechanisms in standard browsers in order to evaluate performance and compatibility costs.

References

[1] 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 2010, 2010. To be published.

[2] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In Proceedings of the 16th ACM Conference on Communications and Computer Security, pages 79–90. ACM Press, 2009.

[3] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In Proceedings of 24th Annual Computer Security Applications Conference, ACSAC '08, pages 322–331. IEEE Computer Society, 2008.

[4] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, pages 109–124. IEEE Computer Society Press, 2010.

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

[6] Martin Johns. On javascript malware and related threats. Journal in Computer Virology, 4:161–178, 2008.

[7] Martin Johns. Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting. PhD thesis, University of Passau, 2009.

[8] Z. Li, K. Zhang, and X. Wang. Mash-IF : Practical Information-Flow Control within Client-side Mashups. In Proceeedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-10), pages 251–260, 2010.

[9] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail : Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In Proceedings of the 19th USENIX Security Symposium, 2010.

[10] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS-10), pages 15–23. ACM Press, 2010.

[11] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In Proceedings of the Symposium on Network and Distributed System Security (NDSS 2007), 2007.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. In IEEE Journal on Selected Areas in Communication, volume 21, pages 5–19, 2003.

[13] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In Proceedings of the 13th ACM Conference on Communications and Computer Security, CCS '06, pages 154–167. ACM Press, 2006.

[14] K. Singh, A. Moshchuk, H.J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, pages 463–478, 2010.