Micro-Policies for Web Session Security

Stefano Calzavara
University Ca' Foscari, Venice
calzavara@dais.unive.it

Riccardo Focardi
University Ca' Foscari, Venice
and Cryptosense, Paris
focardi@unive.it

Niklas Grimm
CISPA, Saarland University
grimm@cs.uni-saarland.de

Matteo Maffei
CISPA, Saarland University
maffei@cs.uni-saarland.de

***Abstract*—Micro-policies, originally proposed to implement hardware-level security monitors, constitute a flexible and general enforcement technique, based on assigning security tags to system components and taking security actions based on dynamic checks over these tags. In this paper, we present the first application of micro-policies to web security, by proposing a core browser model supporting them and studying its effectiveness at securing web sessions. In our view, web session security requirements are expressed in terms of a simple, declarative information flow policy, which is then automatically translated into a micro-policy enforcing it. This leads to a browser-side enforcement mechanism which is elegant, sound and flexible, while being accessible to web developers. We show how a large class of attacks against web sessions can be uniformly and effectively prevented by the adoption of this approach. We also develop a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome: our experiments show that Michrome can be easily configured to enforce strong security policies without breaking the functionality of websites.**

I. Introduction

The Web is nowadays the primary means of access to a plethora of online services with strict security requirements. Electronic health records and online statements of income are a well-established reality as of now, and more and more security-sensitive services are going to be supplied online in the next few years. Despite the critical importance of securing these online services, web applications and, more specifically, *web sessions* are notoriously hard to protect, since they can be attacked at many different layers.

At the network layer, man-in-the-middle attacks can break both the confidentiality and the integrity of web sessions running (at least partially) over HTTP. The standard solution against these attacks is deploying the entire web application over HTTPS with trusted certificates and, possibly, making use of HSTS [19] to prevent subtle attacks like SSL stripping. At the session implementation layer, code injection attacks (or again network attacks) can be exploited to steal authentication cookies and hijack a web session, or to compromise the integrity of the cookie jar and mount dangerous attacks like session fixation [21]. This is particularly problematic because, though the standard HttpOnly and Secure cookie attributes [2] are effective at protecting cookie confidentiality, no effective countermeasure exists as of now to ensure cookie integrity on the Web [36]. Finally, web sessions can also be attacked at the application layer: for instance, since browsers automatically attach cookies set by a website to all the requests sent to it, cross-site request forgery (CSRF) attacks can be mounted by a malicious web page to harm the integrity of the user session with a trusted web application and inject attacker-controlled messages into it. Standard solutions against this problem include the usage of secret tokens and the validation of the Origin header attached by the browser to filter out malicious web requests [3].

In principle, it is possible to achieve a reasonable degree of security for web sessions using the current technologies, but the overall picture still exhibits several important shortcomings and it is far from being satisfactory. First, there are mechanisms like the HttpOnly cookie attribute which are easy to use, popular and effective, but lack flexibility: a cookie may either be HttpOnly or not, hence JavaScript may either be able to access it or be prevented from doing any kind of computation over the cookie value. There is no way, for instance, to let JavaScript access a cookie for legitimate computations, at the cost of disciplining its communication behaviour to prevent the cookie leakage. Then, there are defenses which are sub-optimal and not always easy to implement: this is the case for token-based protection against CSRF. Not only this approach must be directly implemented into the APIs of a web development framework to ensure that it is convenient to use, but also it is not very robust, since it fails in presence of code injection vulnerabilities which disclose the token value to the attacker. Finally, we observe that some attacks and attack vectors against web sessions are underestimated by existing standards and no effective solution against them can be deployed as of now: this is the case for many threats to cookie integrity [36]. These issues will likely be rectified with ad-hoc solutions in future standards, whenever browser vendors and web application developers become more concerned about their importance, and find a proper way to patch them while preserving the compatibility with existing websites.

In this paper, we advocate that a large class of attacks harming the security of web sessions can be provably, uniformly, and effectively prevented by the adoption of *browser-enforced security policies*, reminiscent of a dynamic typing discipline for the browser. In particular, we argue for the adoption of *micro-policies* [16] as a convenient tool to improve the security of web sessions, by disciplining the browser behaviour when interacting with security-sensitive web applications. Roughly, the specification of a micro-policy involves: (1) the definition of a set of *tags*, used to label selected elements of the web ecosystem, like URLs, cookies, network connections, etc., and (2) the definition of a *transfer* function, defining which

operations are permitted by the browser based on the tags and how tags are assigned to browser elements after a successful operation. This kind of security policies has already proved helpful for deploying hardware-level security monitors and nicely fits existing web security solutions, like cookie security attributes [2] and whitelist-based defenses in the spirit of the Content Security Policy [34].

Though previous work has already proposed browser-side security policies as a viable approach for protecting the Web [20], [23], [30], [33], [15], we are the first to carry out a foundational study on a possible extension of a web browser with support for micro-policies and discuss web session security as an important application for this framework. There are many different ways to deploy micro-policies in web browsers, but our proposal is driven by two main design goals aimed at simplifying a large-scale adoption. First, it is *light-weight* and intended to minimize changes to existing web browsers, since it embraces a coarse-grained enforcement approach. Second, it is *practical*: though our proposal is based on a non-trivial theory, we strive for supporting declarative policies for web session security, reminiscent of the tools and the abstractions which web developers already appreciate and use today. We thus propose to express web session security requirements in terms of a simple, declarative information flow policy, which is automatically translated into a micro-policy enforcing it.

To assess the effectiveness of our approach, we developed a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome, and we performed a preliminary experimental evaluation on existing websites. Our experiments show that Michrome can be easily configured to enforce strong security policies without breaking the functionality of websites. We see Michrome as a first reasonable attempt at evaluating the practicality of our theory rather than as a full-fledged defensive mechanism ready for inclusion in standard web browsers. More work is needed to support all the features of our formal model, though we were able to implement and test a significant part of it.

A. Contributions

Our contributions can be summarized as follows:

1) we design FF^τ, a core model of a web browser extended with support for micro-policies. We define the operational behaviour of FF^τ using a small-step reactive semantics in the spirit of previous formal work on browser security [9], [8], [12]. The semantics of FF^τ is parametric with respect to an arbitrary set of tags and the definition of a transfer function operating on these tags;
2) we instantiate the set of tags of FF^τ to intuitive information flow labels and we characterize standard attackers from the web security literature in terms of these labels. We then discuss how to translate simple information flow policies for web session security into micro-policies which enforce them: this is crucial to ensure that most web developers can benefit from our proposal;
3) we discuss example applications of our theory by revisiting known attacks against web sessions and discussing limitations of existing solutions. We then show how these issues are naturally and more effectively solved by our enforcement technique;
4) we develop a prototype implementation of our proposal as a standard Google Chrome extension, Michrome, and we run a set of experiments testing its practicality.

Michrome and a technical report including full proofs are available online [13].

II. Key Ideas

In this section, we give an intuitive overview of the most salient aspects of our framework. We model the browser as a *reactive system*, transforming a stream of input events into a stream of output events. Output events are network requests that are sent by the browser, while input events represent incoming network responses or user actions processable by the browser, e.g., the insertion of a URL into the address bar. Our sets of events include key elements of standard web browsing, like HTTP(S) requests, responses and redirects. For example, the input stream:

$$I = [\mathsf{load}(u), \mathsf{doc_resp}_n(u : \{\mathsf{ck}(k, v)^\ell\}, \mathsf{unit})],$$

instructs the browser to establish a new network connection n to the URL u and retrieve from that connection a response including a cookie $\mathsf{ck}(k, v)^\ell$ and an empty document unit. The cookie, formally seen as a mapping between key k and value v, has a security *label* ℓ, consisting of a confidentiality policy and an integrity policy. For instance, the confidentiality policy $\{\mathsf{https}(d)\}$ expresses that the value of the cookie should only have a visible import for an attacker who is able to decrypt the HTTPS communication with the domain d setting the cookie.

We argue for the adoption of browser-side micro-policies enforcing this kind of security policies. Security is formalized in terms of *reactive non-interference*, a property dictating that similar input streams must always be transformed into similar output streams. Confidentiality is characterized by identifying suitable similarity relations on input streams, based on what the attacker is able to observe about the corresponding output streams. For instance, consider a network attacker with full control of the HTTP traffic: to formalize that cookies with the confidentiality policy $\{\mathsf{https}(d)\}$ have no visible import for the attacker, the stream similarity on inputs may relate streams which are identical except for the value of these cookies.

As an example, let u_s be a HTTPS URL on domain d, it is safe to consider the following two input streams, differing in the cookie value, as similar:

$$I_1 = [\mathsf{load}(u), \mathsf{doc_resp}_n(u : \{\mathsf{ck}(k, v)^\ell\}, \mathsf{unit}), \mathsf{load}(u_s)]$$
$$I_2 = [\mathsf{load}(u), \mathsf{doc_resp}_n(u : \{\mathsf{ck}(k, v')^\ell\}, \mathsf{unit}), \mathsf{load}(u_s)]$$

The reason is that the browser will react to these input streams by producing the following output streams:

$$O_1 = [\mathsf{doc_req}(u : \emptyset), \bullet, \mathsf{doc_req}(u_s : \mathsf{ck}(k, v)^\ell)]$$
$$O_2 = [\mathsf{doc_req}(u : \emptyset), \bullet, \mathsf{doc_req}(u_s : \mathsf{ck}(k, v')^\ell)]$$

These streams include a document request to u without any cookie, a dummy event ($\bullet$) as a reaction to the empty document, and a document request to u_s including the previously

received cookie, which is the normal behaviour of a web browser. Since u_s is a HTTPS URL, the last events of O_1 and O_2 cannot be distinguished by a network attacker, hence the two output streams are similar and there is no violation to reactive non-interference.

But what if the $\mathsf{load}(u_s)$ event in I_1, I_2 was replaced by $\mathsf{load}(u_h)$, where u_h is a HTTP URL on domain d? The behaviour of the browser will be restricted by the underlying micro-policy for non-interference, forcing the production of two output streams not including any cookie in the last event to ensure similarity upon output. These restrictions are enforced by assigning labels to browser components (cookies, connections, scripts...) and by performing runtime label checks upon event processing, reminiscent of a dynamic typing discipline for the browser. Interestingly, simple and intuitive policies like the one we discussed are expressive enough to prevent a large class of known attacks against web sessions. Moreover, despite their simplicity, these policies are actually stronger than currently deployed web solutions (cf. Section VI), providing an expressive mechanism to formally define and enforce confidentiality and integrity properties for web sessions.

III. Background on Reactive Systems

Web browsers can be formalized using labelled transition systems known as *reactive systems* [9], [8]. A reactive system is a state machine which waits for an input, produces outputs in response to it, and repeats the process indefinitely.

Definition 1 (Reactive System)**.** *A* reactive system *is a tuple $R = \langle \mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, C_0, \longrightarrow \rangle$, where $\mathcal{C}$ and $\mathcal{P}$ are disjoint sets of consumer and producer states respectively, while $\mathcal{I}$ and $\mathcal{O}$ are disjoint sets of input and output events respectively. The consumer state C_0 is the initial state of the system and the last component, $\longrightarrow$, is a labelled transition relation over the set of states $\mathcal{Q} \triangleq \mathcal{C} \cup \mathcal{P}$ and the set of events $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$, subject to the following constraints:*

1) *if $C \in \mathcal{C}$ and $C \xrightarrow{a} Q$, then $a \in \mathcal{I}$ and $Q \in \mathcal{P}$;*
2) *if $P \in \mathcal{P}$ and $P \xrightarrow{a} Q$ for some $Q \in \mathcal{Q}$, then $a \in \mathcal{O}$;*
3) *if $C \in \mathcal{C}$ and $i \in \mathcal{I}$, then there exists $P \in \mathcal{P}$ s.t. $C \xrightarrow{i} P$;*
4) *if $P \in \mathcal{P}$, then there exist $o \in \mathcal{O}$ and $Q \in \mathcal{Q}$ s.t. $P \xrightarrow{o} Q$.*

We define *streams* of events through the coinductive interpretation of the following grammar: $S ::= [\,] \mid a :: S$. The semantics of a reactive system R is defined in terms of *traces* (I, O), where I is a stream of input events and O is a stream of output events generated by R as the result of processing I.

Definition 2 (Trace)**.** *Let $R = \langle \mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, C_0, \longrightarrow \rangle$ be a reactive system. Given an input stream I, the state $Q \in \mathcal{C} \cup \mathcal{P}$ generates the output stream O iff the judgement $Q(I) \Downarrow O$ can be coinductively derived by the following inference rules:*

$$\text{(C-NIL)}\ \frac{}{C([\,]) \Downarrow [\,]} \qquad \text{(C-IN)}\ \frac{C \xrightarrow{i} P \quad P(I) \Downarrow O}{C(i :: I) \Downarrow O} \qquad \text{(C-OUT)}\ \frac{P \xrightarrow{o} Q \quad Q(I) \Downarrow O}{P(I) \Downarrow o :: O}$$

We say that R generates the trace *(I, O) iff $C_0(I) \Downarrow O$.*

A natural definition of information security for reactive computations is *reactive non-interference* [9]. We presuppose the existence of a label pre-order $(\mathcal{L}, \sqsubseteq)$ and we represent the attacker as a label $\ell \in \mathcal{L}$, defining its abilities to observe and corrupt data. These abilities are formalized by a label-indexed family of predicates rel_ℓ, identifying security relevant events, and a label-indexed family of similarity relations $\sim_\ell$, identifying indistinguishable events. We collect these two families of relations in a *policy* $\pi = \langle rel_\ell, \sim_\ell \rangle$.

Given a policy π, we define a notion of *similarity* between two streams of events for an attacker ℓ. There are several sensible definitions of similarity in the literature, the one we use here (called *ID-similarity*) leads to a termination-insensitive notion of non-interference and comes with a convenient proof technique based on unwinding relations [9].

Definition 3 (ID-similarity)**.** *Two streams of events S and S' are* ID-similar *(*similar *for short) for ℓ under $\pi = \langle rel_\ell, \sim_\ell \rangle$ iff the judgement $S \approx^\pi_\ell S'$ can be coinductively derived by the following inference rules:*

$$\text{(S-EMPTY)}\ [\,] \approx^\pi_\ell [\,] \qquad \text{(S-MATCH)}\ \frac{rel_\ell(s) \quad rel_\ell(s') \quad s \sim_\ell s' \quad S \approx^\pi_\ell S'}{s :: S \approx^\pi_\ell s' :: S'}$$

$$\text{(S-LEFT)}\ \frac{\neg rel_\ell(s) \quad S \approx^\pi_\ell S'}{s :: S \approx^\pi_\ell S'} \qquad \text{(S-RIGHT)}\ \frac{\neg rel_\ell(s) \quad S \approx^\pi_\ell S'}{S \approx^\pi_\ell s :: S'}$$

Intuitively, a reactive system satisfies non-interference under a policy π if and only if, whenever it is fed two similar input streams, it produces two similar output streams for all the possible attackers (labels).

Definition 4 (Reactive Non-interference)**.** *A reactive system is* non-interferent *under π iff, for all labels ℓ and all its traces (I, O) and (I', O') such that $I \approx^\pi_\ell I'$, we have $O \approx^\pi_\ell O'$.*

Reactive non-interference has been proposed in the past as a useful security baseline to prove protection against common attacks against web sessions, including authentication cookie theft [18], [10], [11] and cross-site request forgery [22].

IV. Micro-Policies for Browser-Side Security

Our model FF^τ is inspired by existing formal models for web browsers based on reactive systems [8], [12]. It is an extension of the Flyweight Firefox model [12] with tags and support for enforcing micro-policies based on them.

A. Syntax

A *map* M is a partial function from keys to values. We let $\{\}$ stand for the empty map and we let $dom(M)$ denote the *domain* of M, i.e., the set of keys bound to a value in M. We let $M_1 \uplus M_2$ be the union of two maps with disjoint domains.

1) Tags: We presuppose the existence of a denumerable set of tags *Tags* and we let τ range over them. We do not put any restriction on the format of these tags, though we instantiate them to a specific format in the next section.

2) Terms: We presuppose a set of domain names $\mathcal{D}$ (ranged over by d) and a set of strings $\mathcal{S}$ (ranged over by s). The *signature* for the set of terms $\mathcal{T}$ is:

$$\Sigma = \{\mathsf{http}, \mathsf{https}, \mathsf{url}(\cdot,\cdot,\cdot), \mathsf{ck}(\cdot,\cdot,\cdot)\} \cup \mathcal{D} \cup \mathcal{S} \cup \mathit{Tags}.$$

Let $\mathcal{X}$ be a set of variables and $\mathcal{N}$ be a set of names, the set of terms $\mathcal{T}$ (ranged over by t) is defined as follows: if $t \in \mathcal{X} \cup \mathcal{N}$, then $t \in \mathcal{T}$; if f is an n-ary function symbol in Σ and $\{t_1, \ldots, t_n\} \subseteq \mathcal{T}$, then $f(t_1, \ldots, t_n) \in \mathcal{T}$.

3) URLs: We let $\mathcal{U} \subseteq \mathcal{T}$ be the set of the URLs, i.e., the set of terms of the form $\mathsf{url}(t, d, s)$ with $t \in \{\mathsf{http}, \mathsf{https}\}$. Given $u = \mathsf{url}(t, d, s)$, let $\mathit{prot}(u) = t$, $\mathit{host}(u) = d$ and $\mathit{path}(u) = s$. We assume that each URL $u \in \mathcal{U}$ comes with an associated tag, returned by a function $\mathit{tag} : \mathcal{U} \rightarrow \mathit{Tags}$. For instance, the *tag* function may assign the Secure tag to HTTPS pages and the Insecure tag to HTTP pages: this information can be used to apply different micro-policies in the browser.

4) Cookies: We let $\mathcal{CK} \subseteq \mathcal{T}$ be the set of cookies, i.e., the set of terms of the form $\mathsf{ck}(s, s', \tau)$. Formally, cookies are just key-value pairs (s, s') extended with a tag τ. We assume this tag is assigned by a function $\kappa : \mathcal{D} \times \mathcal{S} \rightarrow \mathit{Tags}$, so that cookies with the same key set by the same domain must have the same tag. We typically use the more evocative notation $\mathsf{ck}(k, v)^\tau$ to represent cookies. Given $ck = \mathsf{ck}(k, v)^\tau$, we let $\mathit{key}(ck) = k$ and $\mathit{value}(ck) = v$.

5) Scripts: We let values v, expressions e and scripts scr be defined by the following productions:

$$\begin{array}{llll} \textit{Values} & v & ::= & t \mid \mathsf{unit} \mid \lambda x.e \\ \textit{Expr.} & e & ::= & v\,v' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid \mathsf{get\text{-}ck}(v) \\ & & \mid & \mathsf{set\text{-}ck}(v, v') \mid \mathsf{xhr}(v, v') \mid v \\ \textit{Scripts} & \mathit{scr} & ::= & \lceil e \rfloor^{\tau}_{@u} \end{array}$$

A script $\lceil e \rfloor^{\tau}_{@u}$ is an expression e running in the *origin* u with an associated tag τ. The origin u is needed to enforce the same-origin policy on accesses to the cookie jar, while the tag τ is used to enforce micro-policies on the script.

The expression $(\lambda x.e)\, v$ evaluates to $e\{v/x\}$; the expression $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ first evaluates e to a value v and then behaves as $e'\{v/x\}$; the expression $\mathsf{get\text{-}ck}(k)$ returns the value of the cookie with key k, provided that the tag assigned to the cookie allows this operation; the expression $\mathsf{set\text{-}ck}(k, v)$ stores the cookie $\mathsf{ck}(k, v)^\tau$ in the cookie jar, where $\tau = \kappa(\mathit{host}(u), k)$ is a tag derived by the origin u in which the expression is running and the cookie key k; again, the setting operation may fail due to the enforcement of a micro-policy. The expression $\mathsf{xhr}(u, \lambda x.e)$ sends an AJAX request to u and, when a value v is available as a response, it runs $e\{v/x\}$ in the same origin of the script which sent the request. Notably, micro-policies may also be used to constrain AJAX communication in FF^τ. For simplicity, in our model we assimilate to AJAX requests any network request which may be triggered by a script, e.g., the request for an image triggered by the insertion of a markup element in the page where the script is running.

6) Events: Input events i are defined as follows:

$$\begin{array}{lll} i & ::= & \mathsf{load}(u) \\ & \mid & \mathsf{doc_resp}_n(u : CK, e) \mid \mathsf{doc_redir}_n(u : CK, u') \\ & \mid & \mathsf{xhr_resp}_n(u : CK, v) \mid \mathsf{xhr_redir}_n(u : CK, u'). \end{array}$$

The event $\mathsf{load}(u)$ models a user navigating the browser to the URL u: the browser opens a new network connection to u, sends a HTTP(S) request and then waits for a corresponding HTTP(S) response to process over the connection. The event $\mathsf{doc_resp}_n(u : CK, e)$ represents the reception of a document response from u, including a set of cookies CK to set and an expression e to run in the origin u, which leads to the execution of a new script. The event is annotated with the name n of the network connection where the response is received: this connection gets closed when processing the event. The event $\mathsf{doc_redir}_n(u : CK, u')$ models the reception of a HTTP(S) redirection from u to u' along the connection n, setting the set of cookies CK; the event keeps the connection open, while pointing it to u'. A similar intuition applies to XHR responses and redirects. For simplicity, we use $\mathsf{net_resp}_n(u : CK, e)$ to stand for any network response, including redirects.

Output events o are defined as follows:

$$o \quad ::= \quad \bullet \mid \mathsf{doc_req}(u : CK) \mid \mathsf{xhr_req}(u : CK).$$

The event $\bullet$ represents a silent reaction to an input event with no visible side-effect. The event $\mathsf{doc_req}(u : CK)$ models a document request sent to u, including the set of cookies CK. The event $\mathsf{xhr_req}(u : CK)$ models an XHR request sent to u, including the set of cookies CK. We let $\mathsf{net_req}(u : CK)$ represent an arbitrary network request when we do not need to precisely identify its type.

7) States: Browser states are tuples $Q = \langle K, N, H, T, O \rangle$:

$$\begin{array}{llll} \textit{Cookie Jar} & K & ::= & \{\} \mid K \uplus \{d : CK\}, \\ \textit{Connections} & N & ::= & \{\} \mid N \uplus \{n^\tau : u\} \\ \textit{Handlers} & H & ::= & \{\} \mid H \uplus \{n^\tau : (u', \lceil \lambda x.e \rfloor_{@u})\}, \\ \textit{Tasks} & T & ::= & \mathsf{wait} \mid \mathit{scr}, \\ \textit{Outputs} & O & ::= & [\,] \mid o :: O'. \end{array}$$

The cookie jar K maps domain names to the cookies they set in the browser. The network connection store N keeps track of the pending document requests: if $\{n^\tau : u\} \in N$, then the browser is waiting for a document response from u over the connection n. Notice that the network connection includes a tag τ, which makes it possible to enforce micro-policies on it. The handler store H tracks pending XHR requests: if $H(n^\tau) = (u', \lceil \lambda x.e \rfloor_{@u})$, the continuation $\lambda x.e$ is ready to be run in the origin u when an XHR response is received from u' over the connection n. Also these connections have an associated tag.

We use T to represent *tasks*: if $T = \lceil e \rfloor^{\tau}_{@u}$, then a script is running; if $T = \mathsf{wait}$, no script is running. Finally, O is a buffer of output events, needed to interpret FF^τ as a reactive system: let $Q = \langle K, N, H, T, O \rangle$ be a consumer state when $T = \mathsf{wait}$ and $O = [\,]$, otherwise let Q be a producer state. We let $C_0 = \langle \{\}, \{\}, \{\}, \mathsf{wait}, [\,] \rangle$ be the initial state of FF^τ.

B. Reactive Semantics

The reactive semantics of FF^{τ} is parametric with respect to a partial function *transfer* [16], which is roughly a tag-based security monitor operating on the browser model. The transfer function we consider has the following format:

$$\textit{transfer}(\mathsf{event_type}, \tau_1, \tau_2) = (\tau_n, \tau_{ci}, \tau_{co}, \tau_s),$$

where τ_1 and τ_2 are the (at most two) arguments passed to the function when the browser model processes an event of type $\mathsf{event_type}$, while $\tau_n, \tau_{ci}, \tau_{co}, \tau_s$ are the (at most four) tags assigned to the new browser elements which are instantiated as the result of the event processing. Specifically, τ_n is the tag of the new network connection which is created, τ_{ci} is the tag passed to the cookie jar when storing some new cookies, τ_{co} is the tag passed to the cookie jar when retrieving the cookies to be attached to HTTP(S) requests, and τ_s is the tag of the new running script. If any of these elements is not needed when processing an event of a given type, e.g., since no new cookie is set, we replace it with a dash $(-)$. If the transfer function is undefined for a given set of arguments, an operation is not permitted. For space reasons, the full reactive semantics of FF^{τ} is given in the technical report [13]. Here, we just present the main ideas needed to understand the paper.

A set of transitions of the form $C \xrightarrow{i} P$ describes how the consumer state C reacts to the input event i by evolving into a producer state P. Conversely, a set of transitions of the form $P \xrightarrow{o} Q$ describes how a producer state P generates an output event o and evolves into another state Q. Most of the transitions invoke the transfer function before being fired, with the following intuitive semantics:

- $\textit{transfer}(\mathsf{load}, \tau_u, -) = (\tau_n, -, \tau_{co}, -)$: invoked when a URL u such that $\textit{tag}(u) = \tau_u$ is loaded. The event creates a new network connection with tag τ_n and uses tag τ_{co} to access the cookie jar and retrieve the cookies to be attached to the document request sent to u;
- $\textit{transfer}(\mathsf{doc_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)$: invoked when a document response is received over a network connection with tag τ_n. The event uses tag τ_{ci} to access the cookie jar and set the cookies received in the response, while tag τ_s is given to the new script which is executed as the result of processing the response;
- $\textit{transfer}(\mathsf{doc_redir}, \tau_n, \tau_u) = (\tau_m, \tau_{ci}, \tau_{co}, -)$: invoked when a document redirect is received over a network connection with tag τ_n, asking the browser to load a URL u such that $\textit{tag}(u) = \tau_u$. As a result, the tag of the network connection is changed from τ_n to τ_m. The tags τ_{ci} and τ_{co} are used to access the cookie jar: specifically, τ_{ci} is used to set the cookies received along with the processed redirect, while τ_{co} is used to get the cookies to be sent to u upon redirection;
- $\textit{transfer}(\mathsf{xhr_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)$: similar to the case for $\mathsf{doc_resp}$, but for XHR responses;
- $\textit{transfer}(\mathsf{xhr_redir}, \tau_n, \tau_u) = (\tau_m, \tau_{ci}, \tau_{co}, -)$: similar to the case for $\mathsf{doc_redir}$, but for XHR redirects;
- $\textit{transfer}(\mathsf{send}, \tau_s, \tau_u) = (\tau_n, -, \tau_{co}, -)$: invoked when a script with tag τ_s sends an XHR request to a URL u such that $\textit{tag}(u) = \tau_u$. Tag τ_n is given to the new network connection which is opened by the script, while τ_{co} is used to access the cookie jar and get the cookies to be sent to u;
- $\textit{transfer}(\mathsf{get}, \tau_r, \tau_c) = (-, -, -, -)$: invoked when a cookie with tag τ_c is read from the cookie jar. Here, τ_r is the tag modelling the security assumptions about the reader: for instance, when cookies are fetched by the browser for inclusion in a HTTP(S) request to u, this tag could correspond to the protocol of u;
- $\textit{transfer}(\mathsf{set}, \tau_w, \tau_c) = (-, -, -, -)$: invoked when a cookie with tag τ_u is written into the cookie jar. Similarly to the previous case, τ_w is the tag modelling the security assumptions about the writer.

If the transfer function is undefined for a specific set of tags, the corresponding transitions $C \xrightarrow{i} P$ and $P \xrightarrow{o} Q$ just lead to a dummy producer state firing the dummy event $\bullet$.

V. Enforcing Reactive Non-Interference

The operational semantics of FF^{τ} is parametric with respect to an arbitrary set of tags and a transfer function. Here, we instantiate these parameters to show that FF^{τ} can enforce a useful security property, i.e., reactive non-interference.

A. Labels, Policies and Threat Model

We define different threat models for the Web in terms of labels from a pre-order $(\mathcal{L}, \sqsubseteq)$, as required by the definition of reactive non-interference. We start by introducing *simple labels*, which we use to express confidentiality and integrity policies. A simple label l is a (possibly empty) set of elements of the form $\mathsf{http}(d)$ or $\mathsf{https}(d)$ for some domain name $d \in \mathcal{D}$:

$$l ::= \emptyset \mid \{\mathsf{http}(d)\} \mid \{\mathsf{https}(d)\} \mid l \cup l.$$

Intuitively, simple labels define sets of endpoints which are allowed to read/write a given datum or to observe/produce a given event. A *label* $\ell = (l_C, l_I)$ is a pair of simple labels, combining confidentiality and integrity. We write $C(\ell)$ for l_C and $I(\ell)$ for l_I. We let $\ell \sqsubseteq \ell'$ iff $C(\ell) \subseteq C(\ell')$ and $I(\ell) \subseteq I(\ell')$. Simple labels form a bounded lattice under set inclusion, while labels form a bounded lattice under $\sqsubseteq$: the bottom and top elements are $\bot_s = \emptyset$, $\top_s = \{\mathsf{http}(d), \mathsf{https}(d) \mid d \in \mathcal{D}\}$, $\bot = (\bot_s, \bot_s)$ and $\top = (\top_s, \top_s)$.

We assign (simple) labels to URLs, so that it is easy to define which events an attacker can observe and/or corrupt. For a URL $u \in \mathcal{U}$ with $\textit{host}(u) = d$, we let:

- $\textit{msg_label}(u) = \{\mathsf{http}(d)\}$ iff $\textit{prot}(u) = \mathsf{http}$;
- $\textit{msg_label}(u) = \{\mathsf{https}(d)\}$ iff $\textit{prot}(u) = \mathsf{https}$;
- $\textit{evt_label}(u) = \{\mathsf{http}(d)\}$.

We use these functions to define the capabilities of an attacker ℓ. The *presence* of a message sent to u is visible to ℓ whenever $\textit{evt_label}(u) \subseteq C(\ell)$, while the *content* of the message is only disclosed if also $\textit{msg_label}(u) \subseteq C(\ell)$. If $\textit{evt_label}(u) \subseteq C(\ell)$ while $\textit{msg_label}(u) \not\subseteq C(\ell)$, the attacker is aware of the

TABLE I Attacker capabilities for ℓ

Visibility of outputs:

$$\frac{\mathit{evt_label}(u) \subseteq C(\ell)}{\mathit{vis}_\ell(\mathsf{net_req}(u : CK))}$$

Indistinguishability of outputs:

$$\frac{\mathit{msg_label}(u) \not\subseteq C(\ell)}{\mathsf{net_req}(u : CK) \sim_\ell^C \mathsf{net_req}(u : CK')}$$

Taintedness of inputs:

$$\frac{\mathit{msg_label}(u) \subseteq I(\ell)}{\mathit{tnt}_\ell(\mathsf{net_resp}_n(u : CK, e))}$$

presence of all messages sent to u, but he has no access to their contents; the presence of a message may be used to create a side-channel and leak information through an implicit flow. As to integrity, a message coming from u can be *forged* by an attacker ℓ if and only if $\mathit{msg_label}(u) \subseteq I(\ell)$. There is no distinction between message presence and message content when it comes to integrity.

Based on this informal description, the following well-formation hypothesis we make on the attacker should be clear. It ensures that we cannot model attackers who are not aware of the presence of a message, but still have access to its contents.

Definition 5 (Well-formed Attacker)**.** *An attacker* ℓ *is* well-formed *if and only if, for all domains* $d \in \mathcal{D}$*,* $\mathsf{https}(d) \in C(\ell)$ *implies* $\mathsf{http}(d) \in C(\ell)$*.*

From now on, we always implicitly consider only well-formed attackers. It is easy to represent using labels several popular web security attackers:

1) a *web attacker* on domain d is defined by:

$$\ell_w(d) \triangleq (\{\mathsf{http}(d), \mathsf{https}(d)\}, \{\mathsf{http}(d), \mathsf{https}(d)\})$$

2) a *passive network attacker* is defined by:

$$\ell_{pn} \triangleq (\{\mathsf{http}(d) \mid d \in \mathcal{D}\}, \emptyset)$$

3) an *active network attacker* is defined by:

$$\ell_{an} \triangleq (\{\mathsf{http}(d) \mid d \in \mathcal{D}\}, \{\mathsf{http}(d) \mid d \in \mathcal{D}\}).$$

To formalize the previous intuitions, we introduce a few simple ingredients: a *visibility* predicate vis_ℓ on output events, a binary *indistinguishability* relation $\sim_\ell^C$ on output events, and a *taintedness* predicate tnt_ℓ on input events. These are defined in Table I. The indistinguishability relation identifies network requests which only differ for contents (cookies) which are not visible to the attacker, because encrypted. We implicitly assume that two indistinguishable requests have the same type.

We then define two specific classes of non-interference policies, which correctly capture the attacker capabilities we described. Our non-interference results will be restricted to these two classes of policies.

Definition 6 (Confidentiality Policy)**.** *A* confidentiality policy *is a pair* $\pi_C = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ *such that:*

1) $\forall o \in \mathcal{O} : \mathit{rel}_\ell(o) \triangleq \mathit{vis}_\ell(o)$*;*
2) $\forall o, o' \in \mathcal{O} : o \sim_\ell o' \Leftrightarrow o = o' \vee o \sim_\ell^C o'$*.*

Definition 7 (Integrity Policy)**.** *An* integrity policy *is a pair* $\pi_I = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ *such that:*

1) $\forall i \in \mathcal{I} : \mathit{rel}_\ell(i) \triangleq \neg \mathit{tnt}_\ell(i)$*;*
2) $\forall i, i' \in \mathcal{I} : i \sim_\ell i' \Leftrightarrow i = i'$*.*

B. A Canonic Transfer Function for Non-Interference

Our goal is to instantiate the operational semantics of FF^τ with a transfer function that enforces confidentiality and integrity policies. In principle, we could let web developers provide selected entries of the transfer function, defining the browser behaviour upon interaction with their own websites, but this would be quite inconvenient for them. We believe that web developers need a more effective and declarative way to specify their desired confidentiality and integrity policies. In our view, web developers should only:

1) assign security labels to the cookies they set. This is not a hard task, since web developers are already familiar with cookie security attributes like HttpOnly and Secure, and the format of the labels is pretty intuitive;
2) assign security labels to the URLs they control. We argue that also this is not hard to understand for web developers, since this kind of policies is close in spirit to standard Content Security Policy [34] specifications.

These labels define the expected security properties for cookies and network connections:

1) *cookie secrecy*: if a cookie has label ℓ, its value can only be disclosed to an attacker ℓ' such that $C(\ell) \cap C(\ell') \neq \emptyset$;
2) *cookie integrity*: if a cookie has label ℓ, it can only be set or modified by an attacker ℓ' such that $I(\ell) \cap I(\ell') \neq \emptyset$;
3) *session confidentiality*: if a URL u has label ℓ, an attacker ℓ' can observe that the browser is loading u only if we have $C(\ell) \cap C(\ell') \neq \emptyset$;
4) *session integrity*: if a URL u has label ℓ, an attacker ℓ' can force the browser into sending requests to u only if we have $I(\ell) \cap I(\ell') \neq \emptyset$.

Formally, we define a *URL labelling* as a function $\Gamma : \mathcal{U} \to \mathcal{L}$, assigning labels to URLs. If $\Gamma(u) = \ell$ for some label ℓ, let $\Gamma_C(u)$ stand for $C(\ell)$ and $\Gamma_I(u)$ stand for $I(\ell)$. We propose a technique to automatically generate a *canonic* transfer function from a labelling Γ: this function enforces the session confidentiality and integrity properties formalized by Γ, while ensuring that cookies are accessed correctly according to their security label. The canonic transfer function operates on the set of tags $\mathit{Tags} \triangleq \mathcal{L} \cup \mathcal{U}$ including labels and URLs, and assumes that the tagging function for URLs *tag* is the identity on $\mathcal{U}$.

The definition of the canonic transfer function is formalized by using judgements of the following format:

$$\Gamma, f \triangleright \mathit{transfer}(\mathsf{event_type}, \tau_1, \tau_2) \rightsquigarrow \vec{\ell},$$

where Γ, f, $\mathsf{event_type}$ and τ_1, τ_2 are known, while we compute the labels $\vec{\ell}$ to be assigned to the newly created or

updated browser elements when processing an event of type event_type. Here, $f : \mathcal{L} \to \mathcal{L}$ is a *script labelling*, providing a mapping from labels of network connections to labels of scripts downloaded via these connections. Remarkably, while Γ is used to specify different security policies for different URLs and should be provided by web developers, f is just a parameter used to tweak the generation of the canonic transfer function for different use cases. It is useful to have f in the formalism for additional generality, in particular to support the examples in the next section, but in practice (and in our implementation) a good candidate for f is simply the identity function on $\mathcal{L}$. The non-interference results we present hold for any choice of f, as long as it satisfies the following well-formation condition (implicitly assumed from now on).

Definition 8 (Well-formed Script Labelling)**.** *A script labelling f is* well-formed *if and only if, for all labels $\ell \in \mathcal{L}$, we have $C(f(\ell)) \subseteq C(\ell)$ and $I(\ell) \subseteq I(f(\ell))$.*

This well-formation requirement ensures that the confidentiality of a script is always higher than the confidentiality of the network connection from which it is downloaded, while its integrity is always lower than the integrity of the connection. This guarantees that the script cannot disclose the presence of private network connections or trigger high integrity events as the result of an interaction with the attacker.

The judgements defining the canonic transfer function are shown in Table II, using inference rules which should be read as follows: boxed premises amount to checks on Γ, τ_1, τ_2, determining the domain of the transfer function, while premises not included in boxes define the value of the new labels $\vec{\ell}$. If any of the boxed premises fails, the transfer function is undefined and the browser does not process the event. Observe that the necessary entries of the transfer function can be generated "on the fly" upon event processing in an implementation of our theory.

We briefly comment on the rules in Table II as follows. In (G-LOAD), assuming that we load the URL u, we check *evt_label*$(u) \subseteq \Gamma_C(u)$, since a request is sent to u and this request is visible to any network attacker or to any web attacker controlling u. We use $\Gamma_C(u)$ as the confidentiality label of the new network connection to ensure that the presence of that connection in the browser may only be visible to an attacker ℓ such that $\Gamma_C(u) \cap C(\ell) \neq \emptyset$. We use *msg_label*$(u)$ as the integrity label of the new network connection, since an attacker controlling u may be able to compromise the integrity of any response received over that connection. Finally, when accessing the cookie jar to retrieve the cookies to be sent in the request to u, we set *msg_label*(u) as the confidentiality component of the label ℓ_{co} passed to the cookie jar, which ensures that only cookies intended to be disclosed to u will be retrieved. We use $\top_s$ as integrity label for retrieving cookies, so that we get cookies irrespective of their integrity label.

(G-DOCRESP) and (G-XHRRESP) propagate the label ℓ_n of the network connection to the cookie jar when setting new cookies included in a network response received over that connection. In terms of integrity, this implies that a network connection can only set cookies with lower integrity than itself. More subtly, in terms of confidentiality, this also implies that a network connection can only set cookies with higher confidentiality than itself: this is needed to ensure that the attacker cannot detect the occurrence of private network responses from the value (or the existence) of public cookies set in those responses. The rules assign the label $f(\ell_n)$ to the new scripts running after response processing; here, the well-formation of f is crucial to ensure that the confidentiality and integrity restrictions of the script are as least as strong as those of the network connection where they have been downloaded.

In (G-DOCREDIR) and (G-XHRREDIR), when redirecting to a URL u, we check *evt_label*$(u) \subseteq C(\ell_n)$, since a network request is sent to u upon redirection and it may reveal the existence of the network connection. We also have to check $I(\ell_n) \subseteq \Gamma_I(u)$ to ensure that no low integrity connection sends a request to a high integrity URL. We preserve the confidentiality label of the existing network connection, so that the existence of the connection cannot be revealed even after the redirection. Instead, we update the integrity label of the connection to the original integrity label of the connection extended with *msg_label*(u): this formalizes the intuition that the integrity of a network connection gets downgraded through cross-origin redirects. The label used for writing cookies is ℓ_n, just as in (G-DOCRESP), while the label used for fetching cookies is $($*msg_label*$(u), \top_s)$, just as in (G-LOAD).

(G-GET) ensures that no high confidentiality cookie is read by a low confidentiality context and that no low integrity cookie is read by a high integrity context. (G-SET) is the writing counterpart of (G-GET). Finally, (G-SEND) is similar to (G-XHRREDIR), with the role of the incoming network connection taken by a running script (and no cookie set).

C. Reactive Non-Interference

Having defined a canonic transfer function, we now analyze which non-interference properties are supported by it. Let $\mathsf{FF}^{\tau}(\Gamma, f)$ be the instantiation of FF^{τ} with the transfer function derived from Γ and f using the judgements in Table II.

We first discuss confidentiality. The next definition of erasure removes from input events any cookie which must not be visible to the attacker according to its label. By making similar input events that are identical after such erasure, reactive non-interference ensures that the value (and even the presence) of the confidential cookies has no visible import to the attacker.

Definition 9 (Confidentiality Erasure)**.** *Given a set of cookies CK, let ck-erase$_\ell^C(CK)$ be defined as:*

$$\{\mathsf{ck}(s, s')^{\ell'} \in CK \mid C(\ell') \cap C(\ell) \neq \emptyset\}.$$

We then define erase$_\ell^C : \mathcal{I} \to \mathcal{I}$ by applying ck-erase$_\ell^C$ to each CK syntactically occurring in an input event.

The confidentiality theorem combines cookie confidentiality with session confidentiality, i.e., the occurrence of a $\mathsf{load}(u)$ event which must not be visible to the attacker according to the label of u has indeed no visible import on the outputs produced by the browser.

TABLE II Generation of a canonic transfer function from Γ

(G-LOAD)
$$\frac{\boxed{\mathit{evt_label}(u) \subseteq \Gamma_C(u)} \\ \ell_n = (\Gamma_C(u), \mathit{msg_label}(u)) \qquad \ell_{co} = (\mathit{msg_label}(u), \top_s)}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{load}, u, -) \rightsquigarrow (\ell_n, -, \ell_{co}, -)}$$

(G-DOCRESP)
$$\Gamma, f \triangleright \mathit{transfer}(\mathsf{doc_resp}, \ell_n, -) \rightsquigarrow (-, \ell_n, -, f(\ell_n))$$

(G-DOCREDIR)
$$\frac{\boxed{\mathit{evt_label}(u) \subseteq C(\ell_n)} \quad \boxed{I(\ell_n) \subseteq \Gamma_I(u)} \\ \ell_m = (C(\ell_n), I(\ell_n) \cup \mathit{msg_label}(u)) \qquad \ell_{co} = (\mathit{msg_label}(u), \top_s)}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{doc_redir}, \ell_n, u) \rightsquigarrow (\ell_m, \ell_n, \ell_{co}, -)}$$

(G-XHRRESP)
$$\Gamma, f \triangleright \mathit{transfer}(\mathsf{xhr_resp}, \ell_n, -) \rightsquigarrow (-, \ell_n, -, f(\ell_n))$$

(G-XHRREDIR)
$$\frac{\boxed{\mathit{evt_label}(u) \subseteq C(\ell_n)} \quad \boxed{I(\ell_n) \subseteq \Gamma_I(u)} \\ \ell_m = (C(\ell_n), I(\ell_n) \cup \mathit{msg_label}(u)) \qquad \ell_{co} = (\mathit{msg_label}(u), \top_s)}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{xhr_redir}, \ell_n, u) \rightsquigarrow (\ell_m, \ell_n, \ell_{co}, -)}$$

(G-GET)
$$\frac{\boxed{C(\ell_r) \subseteq C(\ell_t)} \quad \boxed{I(\ell_t) \subseteq I(\ell_r)}}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{get}, \ell_r, \ell_t) \rightsquigarrow (-, -, -, -)}$$

(G-SET)
$$\frac{\boxed{C(\ell_t) \subseteq C(\ell_w)} \quad \boxed{I(\ell_w) \subseteq I(\ell_t)}}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{set}, \ell_w, \ell_t) \rightsquigarrow (-, -, -, -)}$$

(G-SEND)
$$\frac{\boxed{\mathit{evt_label}(u) \subseteq C(\ell_s)} \quad \boxed{I(\ell_s) \subseteq \Gamma_I(u)} \\ \ell_n = (C(\ell_s), I(\ell_s) \cup \mathit{msg_label}(u)) \qquad \ell_{co} = (\mathit{msg_label}(u), \top_s)}{\Gamma, f \triangleright \mathit{transfer}(\mathsf{send}, \ell_s, u) \rightsquigarrow (\ell_n, -, \ell_{co}, -)}$$

Theorem 1 (Confidentiality)**.** *Let $\pi_C = \langle rel_\ell, \sim_\ell \rangle$ be the confidentiality policy such that:*

1) $\forall i : \neg rel_\ell(i) \triangleq i = \mathsf{load}(u) \wedge \Gamma_C(u) \cap C(\ell) = \emptyset$;
2) $\forall i, i' : i \sim_\ell i' \Leftrightarrow erase_\ell^C(i) = erase_\ell^C(i')$.

Then, $\mathsf{FF}^\tau(\Gamma, f)$ is non-interferent under π_C.

We now focus on integrity. The next definition of erasure removes from output events any cookie which can be set by the attacker according to its label. By making similar output events that are identical after such erasure, reactive non-interference ensures that only low-integrity cookies can be affected by a manipulation of the input stream performed by the attacker.

Definition 10 (Integrity Erasure)**.** *Given a set of cookies CK, let $ck\text{-}erase_\ell^I(CK)$ be defined as:*

$$\{\mathsf{ck}(s, s')^{\ell'} \in CK \mid I(\ell') \cap I(\ell) = \emptyset\}.$$

We then define $erase_\ell^I : \mathcal{O} \to \mathcal{O}$ by applying $ck\text{-}erase_\ell^I$ to each CK syntactically occurring in an output event.

The integrity theorem combines cookie integrity with session integrity, i.e., the attacker can force the browser into sending network requests to u only if the label of u has a low integrity component.

Theorem 2 (Integrity)**.** *Let $\pi_I = \langle rel_\ell, \sim_\ell \rangle$ be the integrity policy such that:*

1) $\forall o : rel_\ell(o) \triangleq o = \mathsf{net_req}(u : CK) \wedge \Gamma_I(u) \cap I(\ell) = \emptyset$;
2) $\forall o, o' : o \sim_\ell o' \Leftrightarrow erase_\ell^I(o) = erase_\ell^I(o')$.

Then, $\mathsf{FF}^\tau(\Gamma, f)$ is non-interferent under π_I.

D. Proof Sketch

To prove the main theorems in the previous section, we first define a set of syntactic constraints over the structure of the transfer function aimed at enforcing non-interference. One example is the following constraint for load events:

(T-LOAD)
$$\frac{\mathit{evt_label}(u) \cup C(\ell_n) \subseteq \Gamma_C(u) \\ \mathit{msg_label}(u) \subseteq C(\ell_{co}) \qquad \mathit{msg_label}(u) \subseteq I(\ell_n)}{\Gamma \vdash \mathit{transfer}(\mathsf{load}, u, -) = (\ell_n, -, \ell_{co}, -)}$$

Intuitively, rule (T-LOAD) ensures that, when a URL u is loaded, the information $\Gamma_C(u)$ is an upper bound for both $\mathit{evt_label}(u)$ and the confidentiality label $C(\ell_n)$ of the new network connection. Having $\mathit{evt_label}(u) \subseteq \Gamma_C(u)$ implies that the load event is always visible to any network attacker or any web attacker sitting at $host(u)$, while having $C(\ell_n) \subseteq \Gamma_C(u)$ guarantees that the side-effects produced by a response received over the network connection are only visible to $\Gamma_C(u)$. The rule also checks two other conditions: $\mathit{msg_label}(u) \subseteq C(\ell_{co})$ is needed to ensure that the cookies attached to the document request sent to u can actually be disclosed to it, while $\mathit{msg_label}(u) \subseteq I(\ell_n)$ formalizes that an attacker who controls u may be able to compromise the integrity of any response received over the new network connection.

Having defined the full set of constraints, we then use a result from [9] to prove that the FF^τ model satisfies non-interference whenever it deploys a transfer function respecting the constraints.

Definition 11 (Unwinding Relation [9]). *An* unwinding relation *is a label-indexed family of binary relations $\mathcal{R}_\ell$ on states of a reactive system with the following properties:*

1) *if $Q \,\mathcal{R}_\ell\, Q'$, then $Q' \,\mathcal{R}_\ell\, Q$;*
2) *if $C \,\mathcal{R}_\ell\, C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ and $i \sim_\ell i'$ with $rel_\ell(i)$ and $rel_\ell(i')$, then $P \,\mathcal{R}_\ell\, P'$;*
3) *if $C \,\mathcal{R}_\ell\, C'$ and $C \xrightarrow{i} P$ with $\neg rel_\ell(i)$, then $P \,\mathcal{R}_\ell\, C'$;*
4) *if $P \,\mathcal{R}_\ell\, C$ and $P \xrightarrow{o} Q$, then $\neg rel_\ell(o)$ and $Q \,\mathcal{R}_\ell\, C$;*
5) *if $P \,\mathcal{R}_\ell\, P'$, then either of the following conditions hold true:*
   a) *$P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ with $o \sim_\ell o'$ and $Q \,\mathcal{R}_\ell\, Q'$;*
   b) *$P \xrightarrow{o} Q$ with $\neg rel_\ell(o)$ and $Q \,\mathcal{R}_\ell\, P'$;*
   c) *$P' \xrightarrow{o'} Q'$ with $\neg rel_\ell(o')$ and $P \,\mathcal{R}_\ell\, Q'$.*

Theorem 3 ([9]). *Let C_0 be the initial state of a reactive system R. If $C_0 \,\mathcal{R}_\ell\, C_0$ for some unwinding relation $\mathcal{R}$, then R satisfies non-interference.*

We then need to define a suitable unwinding relation to establish non-interference. For confidentiality, we propose a relation that requires equality of the two browsers on the low-confidentiality components, while for integrity we propose a relation that requires equality on high-integrity components. Assuming the aforementioned constraints on the transfer function are respected, we prove that the relations fulfil the conditions of Definition 11 and hence conclude non-interference by Theorem 3. We then show that the canonic transfer function we defined always satisfies the set of constraints, from which we derive our two main theorems.

Note that this approach allows one to syntactically prove non-interference also for transfer functions different from the canonic one, which is a useful and interesting result by itself, as it creates an easy way to show security guarantees of a policy encoded with a custom transfer function. The proofs and the full set of constraints can be found in the online technical report [13].

E. Compatibility and Precision

Another interesting property of the canonic transfer function is that it ensures *compatibility* for websites not implementing the security mechanisms proposed in this paper. Intuitively, it is possible to identify a "weak" URL labelling Γ which does not improve security with respect to standard web browsers, but ensures that no runtime security check performed by the transfer function will ever stop a website from working as originally intended. Formally, we extend the set of output events of FF^τ with a new event $\star$, called *failure*. We then define a variant of FF^τ which is parametric with respect to a URL labelling Γ and explicitly models failures due to the security enforcement performed by the canonic transfer function derived from Γ. This is done by including the event $\star$ in the output stream generated by the reactive system whenever the transfer function is undefined. The failure semantics is presented in the technical report [13].

Let $\Gamma_\top$ be the URL labelling assigning the $\top$ label to each URL, let id be the identity function on labels and let $\mathsf{FF}^\tau_\star(\Gamma_\top, id)$ be the failure-aware variant of FF^τ implementing the canonic transfer function derived from $\Gamma_\top$ and id. We can state the following compatibility theorem.

Theorem 4 (Compatibility). *Let C_0 be the initial state of $\mathsf{FF}^\tau_\star(\Gamma_\top, id)$ and assume that the function $\kappa : \mathcal{D} \times \mathcal{S} \to Tags$ assigns the top label $\top$ to all the elements of its domain. If $C_0(I) \Downarrow O$, then $\star$ does not occur in O.*

While we guarantee the soundness of our approach, we cannot offer perfect *precision*, meaning that our framework conservatively prevents some information flows, even though non-interference is not violated. This is because we do not analyse JavaScript and, consequently, we assume a worst case scenario where information leaks may happen. For example, a script may try to read a confidential cookie and then send an unrelated request to an untrusted domain, which would not break confidentiality. However, without an information flow analysis for JavaScript, this cannot be guaranteed, and thus our approach prevents either the access to the cookie or the network request.

VI. Case Studies

A. Cookie Protection Against Web Attackers

The HttpOnly attribute has been proposed as an in-depth defense mechanism for authentication cookies against web attackers [2]. If a cookie is marked as HttpOnly, the browser forbids any access to it by JavaScript, thus preventing its theft through a successful XSS exploitation. The HttpOnly attribute also provides some integrity guarantees, since JavaScript cannot set or overwrite HttpOnly cookies[1].

Intuitively, a first attempt at representing HttpOnly cookies in our model can be done by giving cookies set by the domain d the label $\ell_c = (\{\mathsf{http}(d), \mathsf{https}(d)\}, \{\mathsf{http}(d), \mathsf{https}(d)\})$, and by ensuring that scripts are assigned the top label $\top$. The label ℓ_c allows the browser to send and set these cookies over both HTTP and HTTPS connections to d. The label $\top$ assigned to scripts, instead, ensures that JavaScript cannot read or write these cookies, as enforced by rules (G-Get) and (G-Set).

As it turns out, however, this labelling forces the implementation of stricter security checks than those performed by standard web browsers on HttpOnly cookies. This is not a limitation of our model, but rather a consequence of the fact that scripts are actually able to compromise the integrity of HttpOnly cookies in current web browsers. Indeed, even if an attacker-controlled script cannot directly set an HttpOnly cookie by accessing the `document.cookie` property, it can still force HttpOnly cookies into the browser by exploiting network communication. For instance, assume that a trusted website `a.com` uses HttpOnly cookies for authentication purposes: a malicious script could run a login CSRF attack by

[1]Though this is not stated explicitly in the cookie specification [2], this is a very sensible security practice and we experimentally verified it on many modern web browsers. It would be easy to model in our framework also HttpOnly cookies which can be set/overwritten by JavaScript, but we preferred to consider the more secure and common behaviour.

submitting the attacker's credentials to `a.com`, thus effectively forcing fresh HttpOnly cookies into the user's browser.

To prevent this class of attacks, the canonic transfer function enforces two further invariants: (1) by rule (G-SEND), all the network connections which are opened by a script are tagged with label $\top$, which enforces that no cookie with integrity label $\{\mathsf{http}(d), \mathsf{https}(d)\}$ can be set over these connections; and (2) by rules (G-DOCREDIR) and (G-XHRREDIR), when a redirect is performed, the integrity label of the network connection receiving the redirect is downgraded to the union of the original integrity label and the message label of the redirect URL. Hence, if a cross-domain redirect is performed, no cookie with integrity label $\{\mathsf{http}(d), \mathsf{https}(d)\}$ can be set over the network connection. This ensures that web attackers cannot exploit malicious scripts or redirects to set cookies with label ℓ_c in the browser, unless they control the domain d.

In the end, standard HttpOnly cookies cannot be accurately modelled in our framework, since the integrity guarantees they provide cannot be expressed by a non-interference policy. Indeed, HttpOnly cookies cannot be set by a script using the `document.cookie` property, but scripts can still set them by exploiting network communication, so it is not clear which label should be assigned to scripts to have non-interference. As we discussed, this asymmetry leaves room for attacks.

Clearly, one can represent in our framework cookies which cannot be read by scripts, but can be set by them, by replacing the cookie label ℓ_c with $\ell'_c = (\{\mathsf{http}(d), \mathsf{https}(d)\}, \top_s)$. These cookies cannot be read by scripts running with the $\top$ label, but they can be liberally set by them. Another plausible design choice would be changing the label given to scripts to let them access cookies labelled ℓ'_c, at the cost of limiting their cross-origin communication. For instance, by giving scripts the label of the connection where they have been downloaded, scripts from the domain d would be allowed to read cookies labelled ℓ'_c, but any cross-domain communication from these scripts will be forbidden by rule (G-SEND) to prevent cookie leakage. This may be a better solution for web applications like e-banking services, which may need to access session state at the client side, but do not interact with untrusted third-parties.

B. Protection Against Gadget Attackers

The gadget attacker has been first introduced in [4] as a realistic threat model for mashup security. A gadget attacker is just a web attacker with an additional capability: a trusted website deliberately embeds a gadget (script) chosen by the attacker as part of its standard functionalities. The embedded gadget may be useful, e.g., for advertisement purposes or for the computation of site-wide popularity metrics. It is well-known that this kind of operation is dangerous on the Web, since the embedded script may be entitled to run in the same origin of the embedding page [28]. For instance, the embedded script may be able to read the authentication cookies of the embedding page. This is largely accepted, however, as long as the author of the embedding page trusts the gadget. But what if the gadget is compromised by the attacker?

Consider a web page hosted at the HTTPS URL u on domain d and loading a gadget from the HTTPS URL u' on domain d'. We can define a labelling Γ such that $\Gamma_C(u) = \{\mathsf{https}(d), \mathsf{https}(d')\}$, which would allow the web page at u to only communicate with HTTPS URLs hosted at d and d' by rule (G-SEND). This may be fine, for instance, if the gadget loaded from u' only computes some local statistics shown in the web page at u. Pick now the following input stream:

$$\begin{aligned} I \;=\; & [\mathsf{load}(u), \\ & \mathsf{doc_resp}_n(u : \{\mathsf{ck}(k, v)^{\ell}\}, \mathsf{xhr}(u', \lambda x.x\ \mathsf{unit})), \\ & \mathsf{xhr_resp}_m(u' : \emptyset, \lambda y.\mathsf{let}\ z = \mathsf{get\text{-}ck}(k)\ \mathsf{in}\ \mathit{leak}(z))] \end{aligned}$$

The input stream I models a scenario where the normally harmless gadget on u' has been somehow compromised by the attacker, so that it will read the value of the cookie k set by the response from u and leak it to the attacker's website, which we assume to be hosted outside the domains d and d'. This attack is prevented by the labelling above, since the XHR request leaking the cookie value is stopped by rule (G-SEND), given that this request would still originate from u.

C. Strengthening PayPal

In 2014 a severe CSRF vulnerability on the online payment system PayPal was disclosed, despite existing server-side protection mechanisms [1]. PayPal employs authentication tokens in order to prevent CSRF attacks, but these tokens could be used multiple times for the same user and, through another vulnerability, it was possible for an attacker to obtain such a valid token for any user. The combination of these two flaws could be used to mount arbitrary CSRF attacks against any PayPal user, e.g., to authorize payments on the user's behalf.

Figure 1 represents a typical payment scenario [29] for a user that has already logged into her PayPal account. The protocol starts with the user clicking on the "buy now" button in an online shop. After being redirected to PayPal, she confirms the payment and the article is successfully purchased. One important detail for the present discussion is that the initial request to PayPal (step 2) is explicitly triggered by the user (step 1) in this scenario. Our goal is to enforce a policy that prevents CSRF attacks against PayPal, while still allowing benign payments. This can be done by setting $\Gamma(u) = (\top_s, \{\mathsf{https}(\texttt{paypal.com})\})$ for all URLs u of PayPal, while letting $\Gamma(u') = \top$ for all URLs u' of the online shop (as we are only interested in protecting PayPal here).

We first explain why this policy does not block benign payments via PayPal, like in Figure 1. Since we have an explicit user action that triggers the initial request to PayPal, we assimilate steps 1-2 of the protocol to the processing of a load event in our formal model. Since $\Gamma_C(u) = \top_s$ for all URLs u of PayPal, the request at step 2 is successfully sent according to rule (G-LOAD). Steps 3-12 of the protocol are always in the domain of PayPal and thus the label used for scripts and network connection is always $(\top_s, \{\mathsf{https}(\texttt{paypal.com})\})$. This allows the browser to perform arbitrary redirects and XHR request to URLs on PayPal, hence all these steps succeed. Finally, since we have $\Gamma_C(u) = \top_s$ for all URLs u

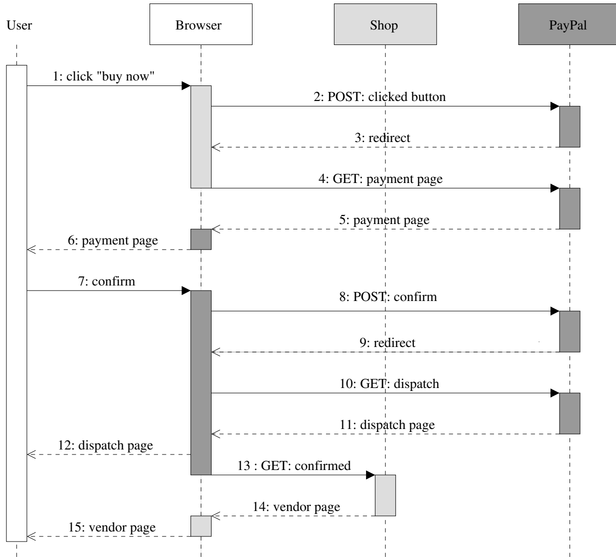

Fig. 1. A typical payment scenario on PayPal

at PayPal and we have $\Gamma_I(u') = \top_s$ for all URLs at the shop domain, also steps 13-15 can be performed successfully, since the cross-origin redirect is permitted by rule (G-DocRedir).

If we now consider a CSRF attack, then we do not have a message that is triggered directly by the user at step 1. This means that we do not have a load event to process, but rather a xhr_req, doc_redir or xhr_redir event coming from a domain different from `paypal.com`. By the definition of the canonic transfer function, we then always have to show either $I(\ell_n) \subseteq \Gamma_I(u)$ or $I(\ell_s) \subseteq \Gamma_I(u)$ in these cases, where u is the URL loaded at step 1. One can then observe that we always have $I(\ell_n) \not\subseteq \{\mathsf{https}(\texttt{paypal.com})\}$ and $I(\ell_s) \not\subseteq \{\mathsf{https}(\texttt{paypal.com})\}$, hence the integrity checks fail and the message is not sent. Thus, our technique effectively prevents the aforementioned CRSF attack against PayPal, even if vulnerabilities are not fixed at the server side.

D. Additional Examples

We also developed two more examples to show our framework at work: cookie protection against network attackers improving on Secure cookies [2] and protection against CSRF in the spirit of Allowed Referrer Lists [15]. For space reasons, these examples are only included in the technical report [13].

VII. Implementation

We developed a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome, which we make available online [13]. We see Michrome as a first reasonable attempt at evaluating the practicality of our theory rather than as a finished product ready for inclusion in standard web browsers. More work is needed to support all the features of FF^τ: for instance, the current prototype lacks support for defining arbitrary cookie labels. We comment in the following on the main implementation choices, our experiments and the current limitations of Michrome.

A. Michrome Implementation

Michrome changes the behaviour of Google Chrome by mimicking the operational semantics of FF^τ, assuming the deployment of the canonic transfer function in Section V-B. The prototype leverages the standard Google Chrome extension APIs, which allow for a rather direct implementation of the semantics. Michrome intercepts web requests via the APIs, and allows or denies them based on the underlying information flow policy. If a blocked request is a navigation to another page, the user receives a message that her request was blocked by the extension. If a blocked request is loading additional content for a page, the user does not receive any specific notification, but she might see differences in the page (e.g., missing images). We discuss below the main differences between Michrome and the presented formal model.

1) User Inputs: FF^τ uniformly treats user inputs as load events. In practice, however, users have different ways to interact with their web browser, most notably by typing in the address bar and by clicking buttons or links. Assimilating all user inputs to load events in Michrome would be a poor design choice, since many of these inputs, e.g., button clicks, can be triggered by malicious JavaScript code, but load events have high integrity in our model. Unfortunately, the Google Chrome extension APIs do not allow one to discriminate between user clicks and clicks performed by JavaScript; similarly, they do not provide any way to distinguish between the user writing in the address bar and a navigation attempt by JavaScript.

Our choice is then to only endorse the first request which is fired from an empty tab and to deem it as the result of a load event, since the only way to trigger a network request from an existing empty tab is by typing in its address bar. All the other network requests are assimilated to less trusted xhr_req events and hence subject to stricter security checks. This policy can be relaxed by defining in Michrome a white-list of trusted entry points, i.e., URLs which are known to be controlled by trusted companies and have a very high assurance of being protected against CSRF attacks: a similar approach has already been advocated in App Isolation [14]. Relaxing the standard behaviour of Michrome is occasionally helpful in practice, for instance to support the PayPal case study (see below).

2) Tagging Scripts: in normal web browsing, many scripts run in the same page (and hence in the same origin) at the same time. The Google Chrome extension APIs do not allow one to detect which script is performing a given operation when more than one script is included in the same page, so Michrome cannot assign labels to individual scripts (unlike the FF^τ model). This issue is solved by giving a label to the entire tab displaying the page rather than to individual scripts. Intuitively, this label represents an upper bound for each label which would be assigned to a script running in the tab.

When a remote content is included from a URL u, the label of the including tab is downgraded and joined with $\Gamma(u)$. There is only one simple exception to this rule: if the included content is passive, i.e., if it is an image, the integrity label of the tab does not get downgraded. This prevents label creeping

for integrity and simplifies the specification of information flow policies for web developers.

3) Policy Granularity: In the current prototype, security labels are assigned to domain names rather than to URLs. This choice is mainly dictated by the practical need of testing our extension on existing websites: having a more coarse-grained security enforcement simplifies the process of writing information flow policies for websites we do not know and control. There is no real mismatch from the formal model here: we just implicitly assume that $\Gamma(u) = \Gamma(u')$ for all u, u' such that $host(u) = host(u')$.

4) Default Behaviour: Formally, the labelling Γ is defined as a (total) function from URLs to labels. In practice, however, one cannot assign a label to all the URLs in the Web. Our choice is to implicitly assume the $\top$ label for all the URLs without an explicit entry in Γ. This solution is suggested by the choice of preserving compatibility with existing websites: since the $\top$ label does not constrain cross-origin communication, the browser behaviour is unchanged when interacting with URLs not included in the labelling.

5) Cookies: At the time of writing, Michrome does not have full support for cookie labels. We plan to implement support for arbitrary labels in the next future, but the current prototype always assumes that a cookie received from a URL u has confidentiality label $\Gamma_C(u)$ and integrity label $\top_s$. This choice is mainly done for the sake of simplicity: by implicitly inferring cookie labels from Γ, we reduce the amount of information which we must specify for the websites we test. The confidentiality label $\Gamma_C(u)$ is justified by the fact that we assume that all URLs on the same domain have the same Γ, hence all the cookies set by them cannot be communicated outside $\Gamma_C(u)$[2]. The integrity label $\top_s$, instead, is motivated by backward compatibility: since standard cookies do not provide good integrity guarantees, we do not try to enforce additional protection in order to avoid breaking websites.

B. Experiments

We performed a first test of Michrome by securing a university website, call it U. Since this website does not include many third-party contents and does not expect to process cross-domain requests, we first assigned U the label: $(\{\mathsf{http}(U), \mathsf{https}(U)\}, \{\mathsf{http}(U), \mathsf{https}(U)\})$. This label states that any session established with U should only be visible to U itself and that only local web pages are allowed to send requests to U. We then realized that this labelling modifies the browser behaviour when navigating U, since the homepage of U silently includes scripts from Google Analytics (*GA*) over HTTP and the extension blocks any request for these scripts, since *GA* should not be aware of the loading of U.

We tried to make Google Analytics work again by adding $\mathsf{http}(GA)$ to the confidentiality label of U. This indeed allowed the browser to send the request for the analytic scripts, but it also prevented the correct rendering and navigation of U later on. The reason is that, when a script from *GA* is included into a page on U, the integrity label of the tab displaying the page is downgraded to include $\mathsf{https}(GA)$. Since the integrity label of U does not mention *GA*, further requests to U from the page are dropped by Michrome: indeed, these requests may be fired by a malicious script mounting a CSRF attack. To recover functionality, we thus had to relax the integrity label of U to also include $\mathsf{https}(GA)$.

Another small change we had to perform to seamlessly navigate U was to extend its confidentiality label to include the sub-domain where the private area of the university is hosted. We also realised the need to include Google (G) in the integrity label of U, otherwise Michrome would prevent the browser from accessing U from the Google search page. Perhaps surprisingly, though Google is entirely deployed over HTTPS, extending the integrity label of U with just $\mathsf{https}(G)$ does not suffice to fully preserve functionality. The reason is that, just like most users, we often omit the protocol and just type `www.google.com` in the address bar to access Google: the browser then tries HTTP by default and then gets automatically redirected to HTTPS by Google. Hence, the integrity label of the tab after the redirect becomes $\{\mathsf{http}(G), \mathsf{https}(G)\}$, which is not good enough to access U. This problem can be solved by using HSTS [19] and preventing any communication attempt to Google over HTTP.

An alternative, simpler solution to this problem is listing the home page of U as a trusted entry point in Michrome, so as to avoid listing all the most popular search engines in the integrity label of U. This is a safe choice in practice, since the homepage of U, like most homepages, is static and does not expect any parameter or untrusted input to sanitize. All in all, we found it pretty easy to come up with an accurate security policy for the website and we think that most web developers should find this process quite intuitive to carry out, especially since this whitelist-based approach is already advocated by existing web standards like Content Security Policy [34].

We also tested Michrome by placing an order on a well-known digital distribution platform, call it D, and by performing the payment using PayPal. We first built an entry for D in the URL labelling, ensuring that both the confidentiality and the integrity components of its label only included D and PayPal. We then set the confidentiality label of PayPal to $\top_s$ and its integrity label to PayPal itself (over HTTPS), thus reconstructing the scenario in the Section VI-C. The payment process worked seamlessly, confirming the result we expected from the formal model.

C. Compatibility and Perceived Performances

Besides the experiments detailed above, we also wrote information flow policies for a small set of national websites and we left Michrome activated in our web browsers while routinely browsing the Web for a few days. We never encountered any visible compatibility issue, even when interacting with websites without an explicit label in the URL labelling, which confirms that the $\top$ label given to them is a sensible default. Clearly, we occasionally broke websites when trying

[2]This is only true if no cookie sharing between sub-domains is possible. Indeed, the current prototype does not protect domain cookies [2], but we plan to include support for them in future releases.

to come up with a correct label to assign to them, but this operation only needs to be done once per website (and only if additional protection is desired for that website). We envision a collaborative effort by security experts and web developers to write down policies for the major security-relevant websites, as it already happens for HTTPS Everywhere [17].

We did not observe any perceivable performance degradation in any of the visited websites, which we do not find surprising, given that the security enforcement ultimately boils down to a few (light-weight) checks on labels.

D. Towards Full Practical Deployment

As we anticipated, Michrome is a proof-of-concept implementation of our approach intended for a first evaluation of its practicality. We implemented Michrome as a browser extension primarily for the sake of simplicity, since the Google Chrome extension APIs are powerful enough to allow us to implement a significant core of our formal framework with limited effort. We are currently investigating whether the entire proposal put forward in this paper can be securely implemented just by using a browser extension. This is not a trivial task, in particular there are (at least) two particularly interesting problems to address. First, implementing support for arbitrary cookie labels would require one to inject wrappers around the getters and setters of the `document.cookie` property. This can be done using a browser extension, but proving the security of the wrappers against arbitrary malicious scripts may be hard: we plan to study existing literature on language-based techniques for isolating JavaScript [6], [24] to address this issue. Moreover, we are investigating to which extent the security guarantees provided by Michrome may hold in presence of other extensions running in the browser: formal browser models representing the extension framework may be useful for the task [5]. Understanding how effectively browser extensions can be employed for improving browser security is an interesting direction in general, since extensions are very easy to deploy and install, hence hold great promise for having a strong practical impact on web security.

VIII. Related Work

Browser-enforced security policies have already been proposed in the past, following two main lines of research. The first research line proposed *purely client-side* defenses like ZAN [32], SessionShield [25], CookiExt [10], [11], CsFire [29] and SessInt [10], which automatically mitigate web applications vulnerabilities by changing the browser behaviour to prevent certain attacks. We improve over these works by giving web developers a tool to express their own browser-enforced security policies, using simple tools and abstractions. This choice makes our proposal more flexible and configurable than previous solutions. We argue that involving web developers in the security process is crucial for the usability and the large-scale deployment of a defensive solution, since purely client-side defenses like the ones we mentioned must implement heuristics to “guess” when their security policy should be applied. These heuristics are bound to (at least occasionally) fail: for instance, CookiExt sometimes breaks the Facebook chat [11], while CsFire prevents certain uses of the OpenId protocol [15].

The second research line on browser-side security, instead, focused on *hybrid* solutions similar to our approach, where the browser enforces a security policy specified by the server [20], [23], [30], [33], [15]. These proposals, however, target very specific attacks like XSS [20], [23], [30] or CSRF [15], rather than providing full-fledged protection for web sessions. Also, these proposals have not been formalized and proved correct. Conversely, in this paper we formalize a rather general micro-policy framework for web browsers and we prove it is expressive enough to support a broad class of useful information flow policies, subsuming existing low-level security mechanisms for web sessions. We think that other useful security properties beyond non-interference can be enforced using micro-policies in the browser: we leave this study for future work.

The present paper was also inspired by previous work on information flow control for web browsers. FlowFox [18] was the first web browser enforcing a sound and precise information flow control on JavaScript by using secure multi-execution. There are many important differences between that approach and the one proposed in this paper. First, FlowFox exclusively prevents attacks posed by malicious scripts, while our proposal covers more common web threats, including malicious HTTP(S) redirects and network attacks. Second, FlowFox does not address integrity threats, though an extension explicitly aimed at thwarting CSRF attacks via scripts has been proposed [22]. Third, FlowFox requires profound changes to the JavaScript engine and has a quite significant impact on browsing performances, while we advocate a much more lightweight approach based on simple checks on labels. This is enough for the web session security properties we target. FlowFox, however, allows the specification of arbitrary fine-grained information flow policies on JavaScript which are beyond the scope of this work.

Fine-grained information flow control for web browsers, and JavaScript in particular, has also been proposed in [7], [27]. These works extend a production JavaScript engine (WebKit) with dynamic information flow control operating at the level of bytecode: [7] presents a first implementation, extended in [27] to account for the intricacies of event handling and the DOM. Both the works come with a soundness proof, establishing termination-insensitive non-interference for the enforcement mechanism. The relative strengths and weaknesses of our proposal with respect to [7], [27] are essentially the same discussed in the comparison between our work and FlowFox. Combining browser-level micro-policies with fine-grained information flow control for JavaScript to provide precise, full-fledged protection for web sessions is an interesting research direction for future work.

Our proposal also shares similar design goals with coarse-grained information flow control frameworks for JavaScript like BFlow [35] and COWL [31]. These frameworks divide scripts in compartments and assign security labels to the latter, to then constrain communication across compartments based

on label checks. An important difference with respect to these works is that the scope of the present paper is not limited to JavaScript. Moreover, we carry out our technical development in a formal model and prove security with respect to this model, while neither BFlow nor COWL have been formalized. Clearly, both BFlow and COWL support the enforcement of general information flow policies on JavaScript code, which is beyond the scope of the present work.

More recently, a research paper reported on the extension of Chromium with support for information flow control based on a lightweight, coarse-grained form of taint tracking [5]. This proposal complements previous work on information flow control for JavaScript by focusing on the entire browser and embracing a wider range of web threats. It might be interesting to explore if the security mechanisms we advocate in this paper can be implemented using the security labels discussed in [5]. The scope of the two works, however, is different: we focus on web session security, while [5] targets intra-browser information flow policies. The threat model in [5] is thus browser-centric, i.e., it identifies attackers with scripts and browser extensions; this is not enough for web session security, an area where network attackers must be taken into account. On the other hand, [5] considers a more detailed browser model than the one used in this paper and it could be a good starting point to extend our work to deal with other threats, e.g., malicious browser extensions. Though we model a smaller fragment of the browser, our approach is intended to require way less changes to existing web browsers than the proposal in [5]: indeed, our framework deliberately targets a good balance between strong web session security guarantees and minimal browser changes to simplify a practical adoption.

Finally, we observe that our label-based policies for confidentiality and integrity are reminiscent of the Same Origin Mutual Approval (SOMA) proposal [26]. SOMA extends the browser with stricter access control checks on content inclusion: both the site operator of the including page and the third party content provider must approve a content inclusion before any communication is allowed by the browser. SOMA is shown to be effective in particular against CSRF attacks and malicious data exfiltration through XSS attacks, which are threats considered also in our work. There are two relevant differences, however, which make our proposal strictly more expressive than SOMA. First and most importantly, SOMA defines an access control mechanism and not an information flow framework: all the security checks performed by SOMA only depend on the including page and the embedded contents, and there is no way to allow or deny a content inclusion based on whether, e.g., the including page has been retrieved by a redirect from the attacker website. Second, SOMA only focuses on network communication and does not support security policies for cookies.

IX. Conclusion

This work explores the usage of micro-policies for the specification and enforcement of confidentiality and integrity properties of web sessions. Micro-policies are specified in terms of tags (here, information flow labels) and a transfer function, which is responsible for monitoring security-relevant operations based on these tags. We modelled the browser as a reactive system and information flow security for web sessions as a non-interference property. We designed a synthesis technique for the transfer function, which allows the end user to specify the expected security policies as simple confidentiality and integrity labels. We demonstrated how our framework uniformly captures a broad spectrum of security policies (e.g., cookie protection, CSRF prevention, and gadget security), improving over existing ad-hoc solutions in terms of soundness and flexibility. We also managed to develop a proof-of-concept implementation of a significant core of our proposal as a simple and efficient Google Chrome extension, Michrome. Our experiments show that Michrome can be configured to enforce strong security policies without breaking the functionality of existing websites.

As future work, we plan to complete the implementation of Michrome to cover the entire framework presented in the paper. We also want to extend our formal model by considering additional browser and webpage components, striving for a good balance between formal expressiveness and ease of deployment in practice. We plan to formalize our development in a theorem prover in order to provide machine-checked security proofs. Furthermore, we would like to design micro-policies tailored to other popular web applications, such as single sign-on protocols, conducting a systematic security analysis of their deployment in the wild.

While performing experiments we realized that Michrome naturally acts as a *learning tool* that collects the integrity level of any web resource that is accessed by a web application. More specifically, when we do not assign security labels to a website, any access is allowed and the integrity level of the browser tab is populated by the security labels of the accessed URLs. This information is useful to have an immediate idea of the "trusted computing base" of the web application and, in many cases, to discover potential vulnerabilities such as importing scripts via HTTP. We plan to complement this learning feature with information about violations of the transfer function, so to automatically derive confidentiality and integrity labels for a whole web application.

Acknowledgements: We thank the anonymous reviewers for their excellent feedback and our shepherd Andrei Sabelfeld for his assistance in improving the final version of the paper. This work was partially supported by the MIUR project ADAPT, by the German research foundation (DFG) through the Emmy Noether program and the collaborative research center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223) - project B4, by the DAAD-MIUR Joint Mobility Program, and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA).

References

[1] Y. Ali, "Hacking paypal accounts with one click (patched)," 2014, available at http://yasserali.com/hacking-paypal-accounts-with-one-click.

[2] A. Barth, “Http state management mechanism,” 2011, available at https://tools.ietf.org/html/rfc6265.

[3] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 75–88.

[4] ——, “Securing frame communication in browsers,” in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, 2008, pp. 17–30.

[5] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, “Run-time monitoring and formal analysis of information flows in chromium,” in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[6] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, “Language-based defenses against untrusted browser origins,” in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 653–670.

[7] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, “Information flow control in webkit’s javascript bytecode,” in *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 159–178.

[8] A. Bohannon and B. C. Pierce, “Featherweight firefox: Formalizing the core of a web browser,” in *USENIX Conference on Web Application Development, WebApps’10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.

[9] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, “Reactive noninterference,” in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, 2009, pp. 79–90.

[10] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, “Automatic and robust client-side protection for cookie-based sessions,” in *Engineering Secure Software and Systems - 6th International Symposium, ESSoS 2014, Munich, Germany, February 26-28, 2014, Proceedings*, 2014, pp. 161–178.

[11] ——, “Cookiext: Patching the browser against session hijacking attacks,” *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[12] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta, “Provably sound browser-based enforcement of web session integrity,” in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, 2014, pp. 366–380.

[13] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei, “Micro-policies for web session security,” 2016, available at https://sites.google.com/site/micropolwebsese.

[14] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, “App isolation: get the security of multiple browsers with just one,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 227–238.

[15] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, “Lightweight server support for browser-based CSRF protection,” in *22nd International World Wide Web Conference, WWW ’13, Rio de Janeiro, Brazil, May 13-17, 2013*, 2013, pp. 273–284.

[16] A. A. de Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, “Micro-policies: Formally verified, tag-based security monitors,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 813–830.

[17] Electronic Frontier Foundation, “HTTPS Everywhere,” 2015, available at https://www.eff.org/https-everywhere.

[18] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “Flowfox: a web browser with flexible and precise information flow control,” in *the ACM Conference on Computer and Communications Security, CCS’12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 748–759.

[19] J. Hodges, C. Jackson, and A. Barth, “Http strict transport security (hsts),” 2012, available at https://tools.ietf.org/html/rfc6797.

[20] T. Jim, N. Swamy, and M. Hicks, “Defeating script injection attacks with browser-enforced embedded policies,” in *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, 2007, pp. 601–610.

[21] M. Johns, B. Braun, M. Schrank, and J. Posegga, “Reliable protection against session fixation attacks,” in *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, 2011, pp. 1531–1537.

[22] W. Khan, S. Calzavara, M. Bugliesi, W. D. Groef, and F. Piessens, “Client side web session integrity as a non-interference property,” in *Information Systems Security - 10th International Conference, ICISS 2014, Hyderabad, India, December 16-20, 2014, Proceedings*, 2014, pp. 89–108.

[23] M. T. Louw and V. N. Venkatakrishnan, “Blueprint: Robust prevention of cross-site scripting attacks for existing browsers,” in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, 2009, pp. 331–346.

[24] S. Maffeis and A. Taly, “Language-based isolation of untrusted javascript,” in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, 2009, pp. 77–91.

[25] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, “Sessionshield: Lightweight protection against session hijacking,” in *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, 2011, pp. 87–100.

[26] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, “SOMA: mutual approval for included content in web pages,” in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 89–98.

[27] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, “Information flow control for event handling and the DOM in web browsers,” in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 366–379.

[28] F. Roesner, T. Kohno, and D. Wetherall, “Detecting and defending against third-party tracking on the web,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 155–168.

[29] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens, “Automatic and precise client-side protection against CSRF attacks,” in *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, 2011, pp. 100–116.

[30] S. Stamm, B. Sterne, and G. Markham, “Reining in the web with content security policy,” in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, 2010, pp. 921–930.

[31] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, “Protecting users by confining javascript with COWL,” in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 131–146.

[32] S. Tang, N. Dautenhahn, and S. T. King, “Fortifying web-based applications automatically,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 615–626.

[33] J. Weinberger, A. Barth, and D. Song, “Towards client-side HTML security policies,” in *6th USENIX Workshop on Hot Topics in Security, HotSec’11, San Francisco, CA, USA, August 9, 2011*, 2011.

[34] M. West, A. Barth, and D. Veditz, “Content security policy (csp),” 2015, available at http://www.w3.org/TR/CSP/.

[35] A. Yip, N. Narula, M. N. Krohn, and R. Morris, “Privacy-preserving browser-side scripting with bflow,” in *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, 2009, pp. 233–246.

[36] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, “Cookies lack integrity: Real-world implications,” in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 707–721.