2014 IEEE 27th Computer Security Foundations Symposium

Provably Sound Browser-Based Enforcement of Web Session Integrity

Michele Bugliesi Stefano Calzavara Riccardo Focardi Wilayat Khan Mauro Tempesta
DAIS, Università Ca' Foscari Venezia
{bugliesi,calzavara,focardi,khan,mtempest}@dais.unive.it

***Abstract*—Enforcing protection at the browser side has recently become a popular approach for securing web authentication. Though interesting, existing attempts in the literature only address specific classes of attacks, and thus fall short of providing robust foundations to reason on web authentication security. In this paper we provide such foundations, by introducing a novel notion of web session integrity, which allows us to capture many existing attacks and spot some new ones. We then propose FF^+, a security-enhanced model of a web browser that provides a full-fledged and provably sound enforcement of web session integrity. We leverage our theory to develop SESSINT, a prototype extension for Google Chrome implementing the security mechanisms formalized in FF^+. SESSINT provides a level of security very close to FF^+, while keeping an eye at usability and user experience.**

I. INTRODUCTION

Despite the growing success of security-critical web applications, "today's Web authentication almost appears to be an exercise in demonstrating how an authentication process should *not* be realized" [28]. Besides its inherent weaknesses, password-based authentication is particularly vulnerable on the Web, since any password entered into a login form flows into the DOM of the page and is made available to any malicious script injected on it. Even when the password is not leaked during the login process, client authentication on the Web is still heavily at risk after the initial authentication step, since a large majority of web applications employ *cookies* to keep track of the authenticated sessions established upon password verification. The attack surface against cookie-based sessions is painfully large: authentication cookies can inadvertently be sent in clear over the wire [26], leaked to malicious websites through XSS flaws [21], or fixated by an attacker [27]. Moreover, untrusted parties may force the browser into creating arbitrary authenticated requests to trusted websites [10]. While current web application frameworks do allow to deploy web authentication safely at the server side, developers often misuse them, and/or are reluctant to adopt recommended security practices [36]. Enforcing protection at the browser side has thus become a popular approach for securing web authentication [15], [18]–[20], [29]–[31], [33]–[35]. Unfortunately, all the existing proposals in the literature only address very specific classes of known vulnerabilities, often lack rigorous security definitions and proofs, and eventually fall short of providing robust foundations for understanding the real effectiveness of client-side defenses for web authentication.

Contributions: In this paper we advocate the study of web authentication security through the introduction of a novel notion of (web) *session integrity*. Our theory draws on *reactive systems*, a formalism which has been previously proposed as an appropriate model of the browser behaviour [13]. Our definition of session integrity is particularly appealing, since it is browser-centric and thus naturally amenable for effective enforcement at the client side, without any background knowledge of the server behaviour. We show how our definition captures many existing attacks and spots some new ones.

We then introduce Flyweight Firefox (FF), a core model of a web browser distilled from the Featherweight Firefox model developed with the Coq proof assistant [11], [12], and we discuss FF^+, a security-enhanced extension of FF that provides a full-fledged enforcement of web session integrity. The runtime mechanisms underlying FF^+ are robust against both web threats and network attacks, and the resulting model is concrete enough to be amenable for an almost direct implementation, while at the same time being fit for a rigorous formal treatment and a security proof.

We leverage our theory to develop SESSINT, a prototype extension for Google Chrome enforcing the security policy formalized in FF^+. SESSINT is a proof of concept that the mechanisms proposed in FF^+ can be implemented in real browsers without affecting too much the user experience of many web applications. In our experiments we identify web scenarios where the security mechanisms of FF^+ need to be relaxed in order to regain usability or functionality of websites: in these cases, SESSINT warns users of the security risk, affecting as less as possible their navigation experience.

DOI 10.1109/CSF.2014.33

Related work: There exists a huge literature on attacks against web authentication, we refer to [28] for a good overview. The research community has proposed several solutions against these attacks in the last few years, based on server-side countermeasures [10], [21], [27], stronger web authentication schemes [4], [17], [23], [25], [28], or purely client-side solutions [15], [18]–[20], [29]–[31], [33]–[35]. In this paper we are particularly interested in the last research line, as browser-side defenses have a very wide scope and applicability: if a website does not comply with recommended security practices and/or is affected by a vulnerability, web authentication can often be protected by working solely at the browser's. Server-side defensive mechanisms or better web authentication protocols are clearly important and worth of study, since they can precisely fix the root cause of the vulnerabilities and prevent usability issues, but we consider these approaches orthogonal to our present endeavours.

We find existing client-side defenses very inspiring and we borrowed (and refined) a number of ideas from them in our work. Still, we observe that different solutions are designed around different threat models, hence it is not obvious how to soundly combine them in practice. We also notice that the lack of formal foundations in previous studies led to the development of sub-optimal solutions: we refer to Section II-A for a subtle attack which can be prevented at the browser side, but escapes state-of-the-art proposals against CSRF.

SessionShield [33] is a client-side proxy aimed at protecting authentication cookies from XSS attacks, by isolating them from JavaScript accesses. The solution protects the confidentiality of authentication cookies against web attacks, but does not enforce protection against network attacks. The same limitation applies to the competitor tool Noxes [30] and to Zan [35].

Several client-side solutions have been proposed against CSRF vulnerabilities [18], [19], [29], [31]. All these tools share the same idea of stripping authentication cookies from (selected classes of) cross-site requests, thus making CSRF attacks largely ineffective. Only the design of [19] has been formally validated, through bounded model-checking. However, the verification excludes from the threat model both XSS flaws and network attackers, which instead are two important aspects we consider in the present work.

Serene [20] is a browser-side solution against session fixation attacks. The core idea is to instruct the browser to attach to outgoing HTTP(S) requests only those authentication cookies which have been set via HTTP(S) headers, thus preventing cookies set by a malicious script from being used for authentication. Serene does not protect against network attacks, since network attackers can overwrite any cookie in the browser just by forging HTTP responses from the registering domain [1], [14]. The design of Serene has not been formally validated.

CookiExt [15] is a recent browser extension aimed at protecting the confidentiality of authentication cookies against both web and network attacks, by marking any authentication cookie received by the browser as both `HttpOnly` and `Secure`, and forcing a redirection from HTTP to HTTPS for supporting websites. The approach has been proved sound through a mechanized non-interference proof, but it does not ensure the *integrity* of authentication cookies and authenticated requests, thus leaving room for attacks like session fixation and CSRF.

A different approach to secure web authentication at the client side would be to extend the browser with a full-fledged information flow control policy, as in FlowFox [24]. At the time of writing, FlowFox does not support integrity policies, which would be central to enforcing our security notion.

Origin cookies have been proposed as a lightweight solution for protecting web sessions, by providing stronger integrity guarantees than standard cookies [14]. Origin isolation is a sound security principle and we leverage it in FF^{+} / SESSINT. However, origin cookies do not solve the problem of protecting the first authentication step, i.e., when the password is sent from the browser to the server. Moreover, origin cookies do not directly support mixed HTTP/HTTPS websites, which instead are largely present on the Web and are supported by our solution (cf. Section IV). We also notice that the `Origin` attribute does not solve all the potential problems affecting cookie-based authentication: for instance, non-`HttpOnly` origin cookies can still be leaked via XSS, so it is not obvious what security guarantees are supported by the `Origin` attribute. On the other hand, origin cookies ensure protection against *related-domain* attackers, which is something we do not consider in our formal model for the sake of simplicity.

A seminal paper by Akhawe et al. [5] proposes a formal definition of web session integrity formulated in Alloy. Roughly speaking, the definition requires that the attacker is not involved in the "causal chain" of the events which lead to an authenticated HTTP(S) request being fired by the browser. The property is very syntactic, so it is hard to generalize it to new settings and carry out a precise comparison with our proposal. What we observe though is that the definition in [5] is only concerned about *web* attackers entering the causal chain: indeed, we argue that it would be difficult to extend the notion to deal with network attackers, since

the latter can enter the causal chain of any transaction which includes at least a communication over HTTP and trivially violate session integrity. Besides the differences in the definitions, we notice that the focus of our work is rather different with respect to [5]: here we target a security property which can be provably enforced at the browser side for *any* authenticated session, with no background knowledge about the intended server behaviour. The authors of [5], instead, use their property to verify some specific browser-server interactions (e.g., the WebAuth protocol) by bounded model-checking.

Similar considerations apply to WebSpi, a ProVerif library for modelling browsers and web applications [9]. While we find the WebSpi approach interesting and general, e.g., it has been applied also to verify cloud-storage services [8], we notice that authenticity properties in ProVerif are modelled through *correspondence assertions*: if we wanted to define web session integrity in these terms, we would need to explicitly model all the pages of the web server and its authentication goals, but this would make it difficult or even impossible to provide integrity guarantees for any authenticated session.

Armando et al. [6], [7] employ formal methods to analyse the security of existing Single Sign-On protocols, exposing real and dangerous attacks against web authentication. The approach is based on bounded model-checking, using SATMC. These papers, however, bear only limited similarities with the present work: their goal is protocol verification and the attacks they report are flaws in the protocol logic, rather than web application vulnerabilities. Their analysis abstracts from many browser-specific and web-specific aspects, which instead are central to the present paper.

Finally, we observe that our focus on client-side defenses has an important impact on the threat model we consider, which is significantly stronger than usual, since we assume that each web page may suffer of both XSS and CSRF. Given that these vulnerabilities are dangerous and widespread in practice, we argue that the design of browser-based defenses like FF^+/ SESSINT should be robust even in the presence of these server-side flaws.

Structure of the paper: Section II introduces our notion of session integrity and shows how it captures different attacks. Section III describes the browser-based enforcement of session integrity in FF^+. Section IV presents our SESSINT implementation. Section V concludes, while the full version [2] provides additional material and proofs.

II. SESSION INTEGRITY

Following [12], we define web browsers in terms of a very general notion of *reactive systems*, based on which we then define session integrity.

Definition 1 (Reactive System)**.** *A* reactive system *is a tuple $(\mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, \longrightarrow)$, where $\mathcal{C}$ and $\mathcal{P}$ are disjoint sets of consumer and producer states respectively, $\mathcal{I}$ and $\mathcal{O}$ are disjoint sets of input and output events respectively. The last component, $\longrightarrow$, is a labelled transition relation over the set of states $\mathcal{S} \triangleq \mathcal{C} \cup \mathcal{P}$ and the set of labels $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$, defined by the following clauses:*

1) *$C \in \mathcal{C}$ and $C \xrightarrow{\alpha} Q$ imply $\alpha \in \mathcal{I}$ and $Q \in \mathcal{P}$;*
2) *$P \in \mathcal{P}$, $Q \in \mathcal{S}$ and $P \xrightarrow{\alpha} Q$ imply $\alpha \in \mathcal{O}$;*
3) *$C \in \mathcal{C}$ and $i \in \mathcal{I}$ imply $\exists P \in \mathcal{P} : C \xrightarrow{i} P$;*
4) *$P \in \mathcal{P}$ implies $\exists o \in \mathcal{O}, \exists Q \in \mathcal{S} : P \xrightarrow{o} Q$.*

A reactive system is an event-driven state machine that waits for an input, produces a sequence of outputs in response, and repeats the process indefinitely without ever getting stuck. We presuppose a lattice of security labels $(\mathcal{L}, \sqsubseteq)$, with bottom and top elements $\bot$ and $\top$. With each output event of a reactive system, we associate a label in $\mathcal{L}$ by way of a *trust* mapping $\tau : \mathcal{O} \to \mathcal{L}$. The intuition is that each label in the lattice corresponds to an interaction point for the reactive system (an *origin*, in the context of web systems), and $\tau(o) = l$ indicates that o is a message output by the reactive system (the browser) in an authenticated session with l's endpoint. We further stipulate that $\tau(o) = \bot$ whenever o does not belong to any authenticated session, and let $\tau_\bot$ stand for the trust mapping such that $\tau_\bot(o) = \bot$ for all $o \in \mathcal{O}$. Finally, we let trust change dynamically, noted $\tau \xrightarrow{o} \tau'$, upon certain output (authentication) events.

Definition 2 (Traces)**.** *Given a trust mapping τ and an input stream I, a reactive system in a state Q generates the output stream O iff the judgement $\tau \vdash Q(I) \rightsquigarrow O$ can be derived by the following inference rules:*

$$\text{(T-NIL)}\ \frac{}{\tau \vdash C([\,]) \rightsquigarrow [\,]} \qquad \text{(T-IN)}\ \frac{C \xrightarrow{i} P \qquad \tau \vdash P(I) \rightsquigarrow O}{\tau \vdash C(i :: I) \rightsquigarrow O}$$

$$\text{(T-OUT)}\ \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau' \vdash Q(I) \rightsquigarrow O}{\tau \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O}$$

A reactive system generates the trace (I, O) *if and only if $\tau_\bot \vdash C_0(I) \rightsquigarrow O$, where C_0 is the initial state of the reactive system.*

Most existing frameworks formalize integrity as a non-interference property predicating that the sensitive (high-level) outputs generated by a system should not depend on the tainted (low-level) information the system receives as an input. This simple idea becomes more complicated in the presence of active attackers, like

the network attackers we consider in this paper. Our proposal is thus reminiscent of *robustness* [22], [32], which intuitively ensures that an active attacker does not have more power than a passive attacker.

We characterize the attacker as a security label $l \in \mathcal{L}$, and define the behaviour of an attacked system in terms of a new output-generation relation $\tau, l, M \vdash Q(I) \rightsquigarrow O$, where M represents the messages the attacker was able to intercept or eavesdrop. The definition is parametric with respect to the relations of interception ($\dagger$), eavesdropping ($?$) and synthesis ($\Vdash$).

Definition 3 (Attacked Traces)**.** *Let l be an attacker. Given an input stream I and a trust mapping τ, an attacked reactive system in a given state Q generates an output stream O (written $\tau, l \vdash Q(I) \rightsquigarrow O$) if and only if the judgement $\tau, l, \emptyset \vdash Q(I) \rightsquigarrow O$ can be derived by the inference rules below:*

$$\text{(AT-NIL)}\ \frac{}{\tau, l, M \vdash C([]) \rightsquigarrow []} \qquad \text{(AT-IN)}\ \frac{C \xrightarrow{i} P \qquad \tau, l, M \vdash P(I) \rightsquigarrow O}{\tau, l, M \vdash C(i :: I) \rightsquigarrow O}$$

$$\text{(AT-OUT)}\ \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau', l, M \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O}$$

$$\text{(AT-GETIN)}\ \frac{\tau, l \dagger i \qquad \tau, l, M \cup \{i\} \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash Q(i :: I) \rightsquigarrow O} \qquad \text{(AT-GETOUT)}\ \frac{P \xrightarrow{o} Q \qquad \tau, l \dagger o \qquad \tau, l, M \cup \{o\} \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash P(I) \rightsquigarrow O}$$

$$\text{(AT-HEARIN)}\ \frac{\tau, l ? i \qquad \tau, l, M \cup \{i\} \vdash Q(i :: I) \rightsquigarrow O}{\tau, l, M \vdash Q(i :: I) \rightsquigarrow O}$$

$$\text{(AT-HEAROUT)}\ \frac{P \xrightarrow{o} Q \qquad \tau \xrightarrow{o} \tau' \qquad \tau, l ? o \qquad \tau', l, M \cup \{o\} \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash P(I) \rightsquigarrow (o, \tau(o)) :: O} \qquad \text{(AT-SYNIN)}\ \frac{C \xrightarrow{i} P \qquad \tau, l, M \Vdash i \qquad \tau, l, M \vdash P(I) \rightsquigarrow O}{\tau, l, M \vdash C(I) \rightsquigarrow O}$$

$$\text{(AT-SYNOUT)}\ \frac{\tau, l, M \Vdash o \qquad \tau \xrightarrow{o} \tau' \qquad \tau', l, M \vdash Q(I) \rightsquigarrow O}{\tau, l, M \vdash Q(I) \rightsquigarrow (o, \tau(o)) :: O}$$

A reactive system generates the attacked trace (l, I, O) *if and only if $\tau_\perp, l \vdash C_0(I) \rightsquigarrow O$, where C_0 is the initial state of the reactive system.*

Our definition of session integrity arises from contrasting the behaviour (i.e., the traces) of a reactive system in the presence, or absence, of an attacker. Given an output stream O, let $O \downarrow l$ denote the stream that results from O by considering only the events at trust level l.

Definition 4 (Session Integrity)**.** *A reactive system preserves* session integrity *for its trace (I, O) iff for all $l \in \mathcal{L}$, and all its attacked traces (l, I, O') one has:*

$$\forall l' \not\sqsubseteq l : O' \downarrow l' \text{ is a prefix of } O \downarrow l'.$$

A reactive system preserves session integrity if and only if it preserves session integrity for all its traces.

Session integrity ensures that the attacker has no effective way to interfere with any authenticated session within the set of traces. In particular, if the trust mapping remains constant at $\tau_\perp$ along the trace, no authentication event occurs in O and the attacker may only initiate its own authenticated sessions, at level l or lower. If instead the trust mapping does change, to include authenticated output events at level $l' \not\sqsubseteq l$, then the requirement that $O' \downarrow l'$ be a prefix of $O \downarrow l'$ ensures that the attacker will at best be able to interrupt the on-going sessions, but not otherwise intrude into them.

A. Web vulnerabilities as session integrity violations

We illustrate a series of attack scenarios, showing how they can be characterized as violations of our session integrity property. We refer to the full version [2] for additional attacks captured by our model, i.e., password theft, login CSRF [10], and session fixation [27].

We picture the attack scenarios as diagrams in which the browser is the reactive system whose input/output events are represented by incoming/outgoing edges respectively. The inputs are generated by the user or correspond to responses from the servers (origins) the browser contacts. The outputs, in turn, are the requests made by the browser or by other origins. Each output is marked by its associated trust level. The diagrams also mark the dynamic changes to the trust mapping along the trace: these arise as a result of authentication events, whose effect is to upgrade the trust level of the cookies set upon authentication to the level of the authentication credentials. The trust level for the credentials is predefined, and given as assumptions *credential* : Origin, where each Origin corresponds to a label in the security lattice. All attack scenarios involve two origins, S and E, placed at incomparable levels in the security lattice: S is the browser's intended partner in the session, while E plays the role of the attacker (or compromised server). The diagrams provide a graphical representation of the attacked traces (cf. Definition 4). The formal encoding of the attacks in the FF model is given in [2].

Cross-Site Request Forgery (Figure 1 (a)): Requested by the user, the browser establishes an authenticated session with S that the server associates with the cookie c: the cookie (the session) assumes a trust label S, based

on the assumption *pwd* : S. Later, the user opens a new page on site E in another browser tab, concluding the unattacked trace. The attacker, sitting at E, provides a response page which automatically triggers a further request to S (via XHR). Being directed to S, for which the browser has registered the cookie c, the new request includes c, thus effectively becoming part of the existing authenticated session with S in the attacked trace. Given that S $\not\sqsubseteq$ E, this violates the prefix condition in our integrity definition.

Reflected XSS (Figure 1 (b)): Like in the previous scenario, the browser establishes an authenticated session with S and associated with a cookie c : S, and later the user requests a new page on site E in another browser tab, concluding the unattacked trace. The response, provided by the attacker at E, redirects the browser to a new page $\hat{u}$ at S, passing a script as a parameter to the page. Assuming S is vulnerable to injection attacks, the script gets included in the response page at $\hat{u}$, which, when rendered, executes the script, thus leaking c to E. At this stage E may generate an output event at level S, which violates the integrity condition for the trace. In the diagram, we tacitly assume that the unattacked part of the trace is over HTTPS, the redirection forced by the attacker is over HTTP, and the cookie c is flagged as `Secure`. If the cookie was not flagged as `Secure`, the attack would resurface as a forgery, like in Figure 1 (a), since c would be attached to the request to $\hat{u}$.

Local CSRF (Figure 1 (c)): This scenario has the same structure as the reflected XSS attack represented in Figure 1 (b). The difference is that the attacker exploits the XSS vulnerability to mount a "same-site" request forgery via the injected script. As a result, unlike the XSS scenario of Figure 1 (b), this attack is effective even when the cookie is flagged as `HttpOnly`. Interestingly, this attack is not prevented by the standard browser-based protection mechanisms against CSRF [18], [19], [29], [31] that strip the cookies from cross-site requests, since the last request is not cross-site.

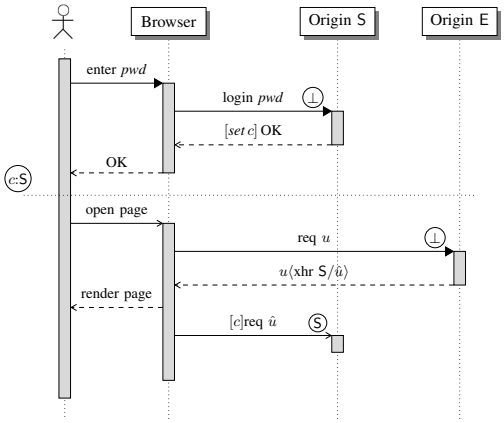
To the best of our knowledge, this last attack is not covered by literature on the subject. Having identified it and devising a technique to guarantee client-side protection against it represent a novel contribution.

III. Enforcing Session Integrity in FF$^+$

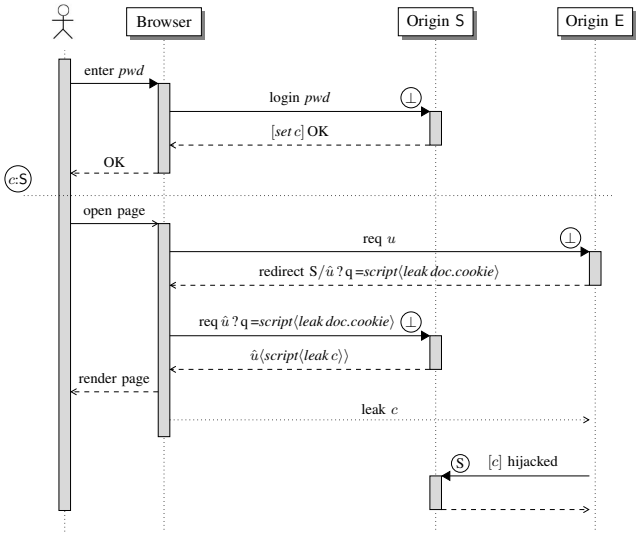
Here we introduce FF, a core model of a standard web browser. We then move from FF to FF$^+$, a security-enhanced variant of FF which enforces session integrity.

A. FF: *syntax and informal semantics*

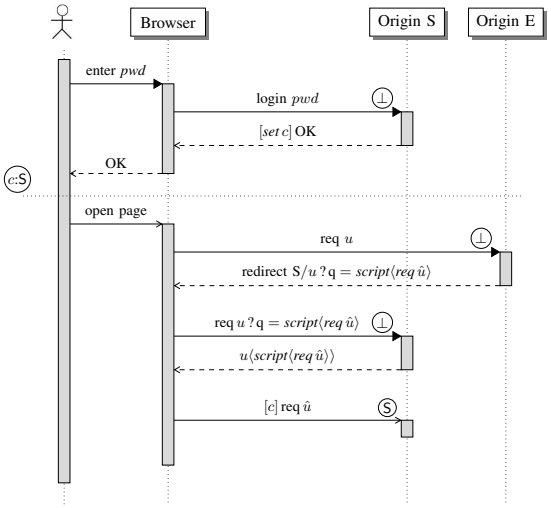
We fix disjoint sets of names $\mathcal{N}$ (a, b, c, d, k, m, n, p) and variables $\mathcal{V}$ (w, x, y, z). A map M is a partial



(a) Classic CSRF



(b) Reflected XSS



(c) Local CSRF

Figure 1: Violations of session integrity (*pwd* : S)

function from keys to values and we write $M(k) = v$ or $\{k \mapsto v\} \in M$ when the key k is bound to the value v in M; $dom(M)$ denotes the domain of M and $\{\}$ is the empty map. Given two maps M_1 and M_2, $M_1 \triangleleft M_2$ is the map M such that $M(k) = v$ iff either $M_2(k) = v$ or $M_1(k) = v$ and $k \notin dom(M_2)$, while $M_1 \uplus M_2$ is the map $M_1 \triangleleft M_2$ whenever $dom(M_1) \cap dom(M_2) = \emptyset$.

1) URLs: We let $\pi \in \{\mathsf{http}, \mathsf{https}\}$ note a protocol identifier. A URL $u \in \mathcal{U}$ is either the constant blank or a triple (π, d, v), where d is a domain name and v is a value encoding additional information, like the full path of the accessed resource or a query string. For $u = (\pi, d, v)$ we let $domain(u) = d$ and $path(u) = v$.

2) Cookies: Cookies are collected in maps ck such that $ck(k) = (n, f)$ whenever the cookie named k is bound to the value n and marked with $f \in \{\mathtt{H}, \mathtt{S}, \top, \bot\}$. Flag `H` models `HttpOnly` cookies, which must not be accessed by JavaScript and only be included in HTTP(S) requests to the registering domain. Flag `S`, in turn, models `Secure` cookies, which must only be sent over encrypted connections. Finally, flag $\bot$ is for cookies with no special security requirements, while $\top$ marks cookies which are both `HttpOnly` and `Secure`. We let $ck\_vals(ck) = \{n \mid \exists k, f : ck(k) = (n, f)\}$.

3) Values and expressions: We let v range over values, i.e., unit, URLs, names, variables and functions:

$$v ::= () \mid u \mid n \mid x \mid \lambda x.e.$$

We let e range over expressions of a simple scripting language which includes first-class functions, basic operations on cookies and the creation of AJAX requests:

$$\begin{array}{rcl} e & ::= & v\ v' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid v? \mid v!\langle v', f\rangle \\ & \mid & \mathsf{xhr}(v, v') \mid \mathsf{auth}(v, v') \mid v. \end{array}$$

$(\lambda x.e)\ v$ evaluates to $e\{v/x\}$; $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ first evaluates e to a value v and then behaves as $e'\{v/x\}$; $k?$ returns the value of cookie k, provided that k is not flagged `HttpOnly`; $k!\langle n, f\rangle$ with $f \in \{\bot, \mathtt{S}\}$ stores the cookie $\{k \mapsto (n, f)\}$ in the cookie jar, ensuring that no existing `HttpOnly` cookie is overwritten. The expression $\mathsf{xhr}(u, \lambda x.e)$ sends an AJAX request to u and, whenever a value v is available as a response, it behaves as $e\{v/x\}$. Finally, $\mathsf{auth}(u, p)$ sends to the URL u the password p.

4) Event handlers: We let h range over (sets of) event handlers, i.e., maps from names to functions. If $h(k) = \lambda x.e$, a handler registered on k is ready to run e, with x bound to the value received along with the firing event. FF handlers model two different aspects of web browsing: first, we use them to encode event-driven JavaScript programming (indeed, we represent the DOM with a set of event handlers). Second, a new handler is instantiated when an AJAX request is sent to a server and it is triggered only when a response is sent back.

5) Pages: Pages are triples $page ::= (u, h, h')$, where u keeps track of the *origin* of the page, h is a set of event handlers registered on the DOM, and h' is a dynamic set of handlers, which grows/shrinks when new AJAX requests/responses are sent/received by the page.

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

$$\begin{array}{rcl} i & ::= & \mathsf{load}(u) \mid \mathsf{text}(p, k, n) \\ & \mid & \mathsf{doc\_resp}(n, ck, u, u', h, e) \\ & \mid & \mathsf{xhr\_resp}(n, ck, u, u', v). \end{array}$$

Event $\mathsf{load}(u)$ models the user navigating the web browser to u: the browser reacts to the event by opening a new network connection to u and sending a request for the document located there. Event $\mathsf{text}(p, k, n)$ corresponds to the user inserting a value n in the text field k of page p: if p contains a set of handlers h such that $h(k) = \lambda x.e$, the event triggers the expression $e\{n/x\}$. Event $\mathsf{doc\_resp}(n, ck, u, \mathsf{blank}, h, e)$ models the receipt of a response from u over the network connection n: the browser will store the cookies ck in its cookie jar, render the document structure (modelled as the set of handlers h) and then run the expression e. Event $\mathsf{doc\_resp}(n, ck, u, u', h, e)$ with $u' \neq \mathsf{blank}$ represents a redirect from u to u': in this case, the cookies ck are stored by the browser, but both h and e are ignored. Event $\mathsf{xhr\_resp}(n, ck, u, \mathsf{blank}, v)$ corresponds to the receipt of an AJAX response from u over the network connection n: the browser will store the cookies ck, then it will retrieve the continuation $\lambda x.e$ which must be triggered by the response, and it will run the expression $e\{v/x\}$. Again, event $\mathsf{xhr\_resp}(n, ck, u, u', v)$ with $u' \neq \mathsf{blank}$ models a redirect from u to u' triggered by an AJAX response (where ck is stored, v is ignored).

Output events o are defined as follows:

$$\begin{array}{rcl} o & ::= & \bullet \mid \mathsf{doc\_req}(ck, u) \mid \mathsf{xhr\_req}(ck, u) \\ & \mid & \mathsf{login}(ck, u, p). \end{array}$$

The dummy event $\bullet$ represents a silent reaction to an input event with no observable side-effect. Event $\mathsf{doc\_req}(ck, u)$ models a document request to u, attaching the cookies ck: it is triggered either by a $\mathsf{load}(u)$ event, or when the browser follows a redirect targeted at u after a document response. Event $\mathsf{xhr\_req}(ck, u)$ models an AJAX request to u, attaching the cookies ck: it is triggered either by the expression $\mathsf{xhr}(u, \lambda x.e)$, or when the browser is redirected to u after an AJAX response. Finally, $\mathsf{login}(ck, u, p)$ represents a request to u which includes the password p, corresponding to the submission of a login form: the occurrence of this event may signal the establishment of a new session. The event

is triggered by the expression $\mathsf{auth}(u, p)$ and it includes the cookies ck which must be sent to u.

We let $\alpha ::= i \mid o$ range uniformly over input and output events. We refer to requests, responses and logins as *network* events.

7) Browser states: Browser states are 5-tuples $Q = \langle W, K, N, T, O\rangle$ where:

$$\begin{aligned} \textit{Windows } W &::= \{\} \mid \{p \mapsto \textit{page}\} \mid W \uplus W, \\ \textit{Cookies } K &::= \{\} \mid \{d \mapsto ck\} \mid K \uplus K, \\ \textit{Networks } N &::= \{\} \mid \{n \mapsto (u, v)\} \mid N \uplus N, \\ \textit{Tasks } T &::= \{\} \mid \{p \mapsto e\}, \\ \textit{Outputs } O &::= [\,] \mid o. \end{aligned}$$

The window store W maps fresh page identifiers to pages, while the cookie jar K maps domain names to the cookies they registered in the browser. The network connection store N keeps track of the open network connections: if $\{n \mapsto (u, v)\} \in N$, then the browser is waiting for a document/AJAX response from u (the role of v will be apparent in the formal semantics). We use T to represent *tasks*: if $\{p \mapsto e\} \in T$, then the expression e is running in the page p. Finally, O is a size-1 buffer of output events, which is convenient to interpret our model as a reactive system.

We say that $Q = \langle W, K, N, T, O\rangle$ is a *consumer* state when both T and O are empty and we denote it with C, otherwise we say that Q is a *producer* state and we denote it with P. The formal semantics of FF is given in the full version [2].

B. Session establishment

Our definition of session integrity relies on a lattice of security labels, which we instantiate next.

Definition 5 (Security Labels)**.** *The set of* security labels *$\mathcal{L}$, ranged over by l, is the smallest set generated by the following grammar:*

$$l := \bot \mid \top \mid \mathsf{evil} \mid \mathsf{net} \mid \pi(d) \textit{ with } \pi \in \{\mathsf{http}, \mathsf{https}\}.$$

We define $\sqsubseteq$ as the least pre-order over $\mathcal{L}$ with $\bot$ as a bottom element, $\top$ as a top element, induced by the axioms: $\{\mathsf{evil} \sqsubseteq \mathsf{http}(d), \mathsf{http}(d) \sqsubseteq \mathsf{net}, \mathsf{net} \sqsubseteq \mathsf{https}(d)\}$.

We assume a partial function $\mathit{url\_label} : \mathcal{U} \to \mathcal{L}$ such that $\mathit{url\_label}(u) = \pi(d)$ whenever $u = (\pi, d, v)$. We also stipulate that the set of names $\mathcal{N}$ is partitioned into the indexed family $\{\mathcal{N}_l\}_{l \in \mathcal{L}}$: this is needed to capture in the model the inability of the attacker to guess random secrets, like passwords or authentication cookie values.

We adopt password-based authentication to establish new sessions with remote web servers. Simply put, when a valid password is submitted to a website supporting authenticated access, a cookie is endorsed to identify the password's owner for the session. Formally, this amounts to instantiating the relation $\tau \xrightarrow{o} \tau'$ underlying the semantics of reactive systems (cf. Definition 2). For this purpose, we presuppose a function $\rho : \mathcal{N} \to \mathcal{L}$ with the following understanding: if $\rho(n) = \pi(d)$, then n is the user's password for the website at d and can be exchanged on the protocol π. We let $\rho(n) = \mathsf{evil}$ whenever n is a password identifying the attacker's account: for simplicity, we assume that this password can be used to establish authenticated sessions on any website. We assume ρ to be *consistent* with respect to the partitioning of names, i.e., we stipulate $\rho(n) \sqsubseteq l$ whenever $n \in \mathcal{N}_l$.

Let now $\mathcal{U}_{\mathit{auth}} \subseteq \mathcal{U}$ be the set of the URLs containing a login form for password-based authentication. We assume that $\mathcal{U}_{\mathit{auth}}$ is partitioned into two subsets $\mathcal{U}_{\mathit{ok}}$ and $\mathcal{U}_{\mathit{fix}}$. If a valid password c is sent to $u \in \mathcal{U}_{\mathit{ok}}$, a fresh authentication cookie is created by the server and employed to identify the password's owner; if $u \in \mathcal{U}_{\mathit{fix}}$, instead, the server may be subject to session fixation, hence it endorses for authentication a cookie already included in the login request. In both cases the (only) authentication cookie is chosen by a function $\kappa : \mathcal{U}_{\mathit{auth}} \to \mathcal{N}$ identifying its name, and the trust mapping is updated to reflect that any output event o including that cookie will have the trust level $\rho(c)$ bound to the password, much like in the examples of Section II-A. The formal details are in Table 1 and commented below.

Rule (A-SRV) models a login on $u \in \mathcal{U}_{\mathit{ok}}$. If c is a valid password, a fresh value n is picked from the name partition $\mathcal{N}_{\rho(c)}$ based on an underlying total order ($n \leftarrow \mathcal{N}_{\rho(c)}$). The value n will be used to identify the password's owner: specifically, we perform a pointwise join between the original trust function τ and the auxiliary trust function $\tau_{u,n,c}$ in the table, which raises to $\rho(c)$ the trust of the output events sent to $\mathit{domain}(u)$ which include the cookie $\{\kappa(u) \mapsto (n, f)\}$ for some f.

Rule (A-FIX), instead, models a login on $u \in \mathcal{U}_{\mathit{fix}}$. In this case, the value n bound to the key $k = \kappa(u)$ among the cookies ck sent to the server will be used to identify the password's owner. If $k \notin \mathit{dom}(ck)$, the authentication fails and rule (A-NIL) must be applied.

C. Threat model

We assume that all HTTPS traffic is signed using trusted certificates (unsigned HTTPS traffic is represented using HTTP). The attacker's power is characterized by a security label l, with the understanding that higher labels provide additional capabilities. A novel aspect of our threat model is that we assume the attacker has full control over *compromised* sessions, i.e.,

TABLE 1 Rules for password-based authentication

$$\text{(A-SRV)}\ \frac{u \in \mathcal{U}_{ok} \qquad n \leftarrow \mathcal{N}_{\rho(c)} \qquad \rho(c) \in \{url\_label(u), \mathsf{evil}\}}{\tau \xrightarrow{\mathsf{login}(ck,u,c)} \tau \sqcup \tau_{u,n,c}}$$

$$\text{(A-FIX)}\ \frac{u \in \mathcal{U}_{fix} \qquad \kappa(u) = k \qquad ck(k) = (n, f) \qquad \rho(c) \in \{url\_label(u), \mathsf{evil}\}}{\tau \xrightarrow{\mathsf{login}(ck,u,c)} \tau \sqcup \tau_{u,n,c}}$$

$$\text{(A-NIL)}\ \frac{\alpha \text{ has a different form}}{\tau \xrightarrow{\alpha} \tau}$$

where $\tau_{u,n,c}(o) = \begin{cases} \rho(c) & \text{if } o \in \{\{\mathsf{doc}, \mathsf{xhr}\}\_\mathsf{req}(ck', u') \mid domain(u) = domain(u') \wedge ck'(\kappa(u)) = n \wedge \tau(o) \sqsubseteq \rho(c)\} \\ \bot & \text{otherwise} \end{cases}$

authenticated sessions established using the attacker's credentials. If a network request belongs to a compromised session, we pessimistically assume that all the data included in the request are stored by the server in the attacker's account and later made available to him: this is useful to capture login CSRF attacks [10].

Formally, the threat model results from instantiating the definitions of interception (†), eavesdropping (?) and synthesis ($\Vdash$) in Definition 2. Let $ev\_label : \mathcal{A} \to \mathcal{L}$ be the function such that $ev\_label(\alpha) = url\_label(u)$ whenever α is a network event sent to/received from u; we assume $ev\_label(\alpha) = \top$ whenever α is not a network event.

The relations † and ? are defined as follows:

$$\text{(II-NET)}\ \frac{ev\_label(\alpha) \sqsubseteq l}{\tau, l \dagger \alpha} \qquad \text{(IH-NET)}\ \frac{ev\_label(\alpha) \sqcap \mathsf{net} \sqsubseteq l}{\tau, l \,?\, \alpha} \qquad \text{(IH-EVIL)}\ \frac{\tau(o) = \mathsf{evil}}{\tau, l \,?\, o}$$

According to rule (II-NET), a web attacker at level, say, $\mathsf{http}(d)$ can intercept only the network traffic sent to d either in clear or with no trusted certificates, while a network attacker can intercept all the HTTP traffic (and any HTTPS message directed to him). We remark that a net-level attacker cannot intercept arbitrary HTTPS traffic: indeed, since signed HTTPS communication ensures both freshness and integrity [3], the attacker cannot replay encrypted messages or otherwise tamper with HTTPS exchanges without breaking the communication session. Hence, preventing the interception of arbitrary HTTPS traffic ultimately amounts just to discarding denial of service attacks, which we are not interested to deal with in the present paper. Notice, however, that an HTTPS exchange can still be overheard by a net-level attacker using rule (IH-NET): network attackers are thus aware of all the network traffic, even though they may be unable to access its payload. Finally, rule (IH-EVIL) makes any request sent over compromised sessions available to the attacker, as we discussed above.

Defining the relation $\tau, l, M \Vdash \alpha$ is slightly more complex. We start by defining an auxiliary relation $\tau, l, M \Vdash n$, which identifies the names that can be generated by the attacker:

$$\text{(NS-BASE)}\ \frac{n \in \mathcal{N}_{l'} \qquad l' \sqsubseteq l}{\tau, l, M \Vdash n} \qquad \text{(NS-LOOK)}\ \frac{ev\_label(\alpha) \sqsubseteq l \qquad n \in fn(\alpha)}{\tau, l, M \cup \{\alpha\} \Vdash n} \qquad \text{(NS-EVIL)}\ \frac{\tau(o) = \mathsf{evil} \qquad n \in fn(o)}{\tau, l, M \cup \{o\} \Vdash n}$$

According to (NS-BASE), an l-attacker can generate any name in a name partition indexed by a label bounded above by l. By rule (NS-LOOK) the attacker may generate the free names of any network event α previously intercepted or overheard, provided that the attacker can inspect its payload. Finally, rule (NS-EVIL) grants the attacker the capability to generate any name communicated over compromised sessions.

Now, we can define the relation $\tau, l, M \Vdash \alpha$:

$$\text{(IS-GEN)}\ \frac{\alpha = i \Rightarrow ev\_label(\alpha) \sqsubseteq l \qquad \forall n \in fn(\alpha) : \tau, l, M \Vdash n}{\tau, l, M \Vdash \alpha} \qquad \text{(IS-REP)}\ \frac{\alpha \in M \qquad ev\_label(\alpha) \sqsubseteq \mathsf{net} \sqsubseteq l}{\tau, l, M \Vdash \alpha}$$

By rule (IS-GEN), an l-attacker can forge an input event i, provided that he can generate all the free names in i and the event label of i is bounded above by l: the latter condition ensures, for instance, that a net-level attacker cannot forge signed HTTPS traffic and that a web attacker $\mathsf{http}(d)$ cannot provide responses for another web server at d'. Rule (IS-GEN) also allows the attacker to send arbitrary output events to any server, provided that he is able to compose the request contents. Finally, rule (IS-REP) allows an attacker with network capabilities (side-condition $\mathsf{net} \sqsubseteq l$) to replay previously intercepted/ overheard traffic. Since HTTPS ensures freshness, the side-condition $ev\_label(\alpha) \sqsubseteq \mathsf{net}$ similarly guarantees that encrypted traffic cannot be replayed.

We conclude this section with a note on XSS attacks: we implicitly include them in our model, since our session integrity property quantifies over all the possible inputs made available to the browser. This universal quantification grants any attacker the capability to mount reflected XSS attacks on any website.

D. FF$^+$: a secure extension of FF

FF provides a faithful abstraction of current web browsers and, just like them, it is vulnerable to a variety of attacks. In this section, we discuss the design of FF$^+$, a security-enhanced extension of FF aimed at enforcing web session integrity.

1) Qualifiers: FF lacks the contextual information needed to apply a sound security policy for session integrity, since it does not track origin changes across network requests. We fix this by extending the structure of network connections and pages with *qualifiers*, by having $N ::= \{\} \mid \{n \mapsto (u, v, q)\} \mid N \uplus N$ and *page* $::= (u, h, h', q)$. A qualifier $q \in \{\checkmark, \times\}$ is just a boolean mark used to *taint track* the open network connections. Pages downloaded from a given connection inherit the qualifier assigned to the connection, and connections become tainted when a cross-origin redirect is performed over them. FF$^+$ enforces different security policies on a page based on the value of its qualifier.

2) Security contexts: FF must also be enhanced to prevent the risk of password theft. When the user enters a password into a login form, an event handler registered on the page can steal the password and leak it to the attacker. We address this issue by running each expression e inside a *security context*, i.e., a sandbox represented by a pair (e, l). If $l = \pi(d)$, then the expression e is allowed to communicate only with d on the protocol π. When a password n is disclosed to an expression e, we instantiate a new security context $(e, \rho(n))$, which provides FF$^+$ with the information needed to protect n. Clearly, this assumes that FF$^+$ keeps track of $\rho(n)$ for any password n input by the user, for instance by using an internal password manager[1]. Formally, we enrich the syntax of tasks by having $T ::= \{\} \mid \{p \mapsto (e, l)\}$.

3) Secure cookie operations: Updates to the cookie jar in FF$^+$ adopt a strong security policy, whereby authentication cookies received over HTTP are marked `HttpOnly`, while authentication cookies received over HTTPS are flagged both `HttpOnly` and `Secure`. If a `Secure` cookie is sent from the server to the browser over HTTP, which is one of the many quirks allowed on the Web, it is discarded by FF$^+$. Moreover, FF$^+$ strengthens the integrity of cookies set over HTTPS against network attacks, by ensuring that cookies which are marked as both `HttpOnly` and `Secure` are never overwritten by cookies set through HTTP responses. This is not ensured by standard web browsers [1] and previous proposals already highlighted the dangers connected to this practice [14]. The formal details correspond to the secure cookie update function *sec_upd_ck* in Table 2.

We also introduce a secure counterpart of the standard procedure employed by web browsers to select the cookies to be attached to a given network request. Specifically, FF$^+$ ensures that no outgoing cookie can have been fixated by an attacker: for HTTP requests we enforce protection against web attacks, by requiring that only `HttpOnly` cookies are sent to the web server. Since these cookies cannot be set by a script, they can only be fixated by network attacks. For HTTPS requests, instead, we target a higher level of protection and we ensure that any cookie attached to them cannot have been fixated, even by a network attacker. Accordingly with the previous discussion, we thus impose that only cookies which are marked as both `Secure` and `HttpOnly` are attached to HTTPS requests. The formal details amount to the definition of the function *get_http_ck* in Table 2.

4) Inputs: The transitions $C \xrightarrow{i} P$ in Table 3 describe how the consumer state C reacts to the input i by evolving into a producer state P. The definition of $C \xrightarrow{i} P$ consists only of two rules, i.e., (I-MIRROR) and (I-COMPLETE). The definition relies on the auxiliary relation $C \overset{i}{\mapsto} P$, which is the bulk of the semantics: this is convenient to interpret FF$^+$ as a reactive system.

We start with the behaviour of document requests and responses. When the user navigates the browser to a URL u, a new network connection n is created and it is assigned the qualifier ✓ by rule (I-LOAD). Moreover, a new document request event is generated and put in the output buffer. If a cross-origin redirect is received over n, the connection is given the qualifier × and becomes tainted, and it will never be restored to an untainted state by rule (I-DOCREDIR). Further requests sent over a tainted connection n will never include cookies, to thwart CSRF attacks performed through a redirect: this policy is applied also to same-origin requests, to prevent local CSRF attacks similar to the one described in Section II-A. When a document response is eventually received over the network connection n, the connection is closed and a new page is stored in the browser by rule (I-DOCRESP). The page inherits the qualifier assigned to n and the cookie jar is updated only if n was marked as untainted: this is needed to prevent the attacker from corrupting the cookies stored in the browser through malicious redirects.

Text input events are handled by rule (I-TEXT) as anticipated, by letting the disclosed expression $e\{n/x\}$ run in the security context $\rho(n)$, which will ensure that the confidentiality of passwords is protected. Finally, AJAX responses are processed much like document re-

[1]For simplicity, in the formal model each password is associated to a single origin. Our implementation allows to reuse the same password on different websites (cf. Section IV).

TABLE 2 Secure management of the cookie jar

$$\frac{}{\{\} \nearrow \pi = \{\}} \qquad \frac{f \in \{\bot, \mathtt{H}\}}{\{k \mapsto (n,f)\} \nearrow \mathsf{http} = \{k \mapsto (n,\mathtt{H})\}} \qquad \frac{f \in \{\mathtt{S}, \top\}}{\{k \mapsto (n,f)\} \nearrow \mathsf{http} = \{\}}$$

$$\frac{}{\{k \mapsto (n,f)\} \nearrow \mathsf{https} = \{k \mapsto (n,\top)\}} \qquad \frac{ck_1 \nearrow \pi = ck_1' \qquad ck_2 \nearrow \pi = ck_2'}{(ck_1 \uplus ck_2) \nearrow \pi = ck_1' \uplus ck_2'}$$

For $u = (\pi, d, v)$, we let:

$$\mathit{sec\_upd\_ck}(K,u,ck) = \begin{cases} K \uplus \{d \mapsto (ck \nearrow \pi)\} & \text{if } d \notin \mathit{dom}(K) \\ K' \uplus \{d \mapsto (ck' \lhd (ck \nearrow \pi))\} & \text{if } K = K' \uplus \{d \mapsto ck'\} \wedge \pi = \mathsf{https} \\ K' \uplus \{d \mapsto (ck_h \lhd (ck \nearrow \pi \lhd ck_s))\} & \text{if } K = K' \uplus \{d \mapsto ck_h \uplus ck_s\} \wedge \pi = \mathsf{http} \end{cases}$$

where:

$$\forall k \in \mathit{dom}(ck_h) : ck_h(k) = (n,f) \Rightarrow f \in \{\bot, \mathtt{H}, \mathtt{S}\}$$
$$\forall k \in \mathit{dom}(ck_s) : ck_s(k) = (n,f) \Rightarrow f = \top.$$

Finally, we let $\mathit{get\_http\_ck}(K,u)$ be defined as the least map M such that:

$$M(k) = \begin{cases} (n,\top) & \text{if } u = (\mathsf{https}, d, v) \wedge \exists ck : K(d) = ck \wedge ck(k) = (n,\top) \\ (n,\mathtt{H}) & \text{if } u = (\mathsf{http}, d, v) \wedge \exists ck : K(d) = ck \wedge ck(k) = (n,\mathtt{H}) \end{cases}$$

sponses. The only interesting differences from a security perspective are in rule (I-XHRRESP), where we must additionally instantiate the label of the new security context to the url_label of the page which sent the AJAX request: this is needed to protect the confidentiality of passwords when the continuation of an AJAX request is executed. It is also worth noticing that we require the qualifier q of the network connection to match the qualifier of the page where the response is received: loading tainted scripts inside an untainted page would be unsound, since these scripts would be allowed to send authenticated requests (see below).

5) Outputs: Table 4 collects the transitions $P \xrightarrow{o} Q$, describing how a producer state P can generate an output o and evolve into another state Q. Several rules are standard, so we just comment the most interesting points.

Rule (O-SET) models the setting of a cookie via JavaScript. The upd_ck function stands for the standard cookie update operation available in web browsers (formalized in [2]). The only point worth mentioning here is the security label $\bot$ required on the security context, which is needed to prevent confidentiality leaks resulting by setting a cookie containing password information.

Rule (O-XHR) is the most complex and models the sending of an AJAX request by an expression running on a page. Different security policies are applied, based on the qualifier of the page and the label of the security context where the expression is run. Let u' be the URL of the page, q the qualifier of the page, l the label of the security context, and u the destination of the AJAX request. When $l = \bot$, no password was previously typed by the user and no confidentiality policy is enforced; otherwise, FF^+ allows the sending of the request only if $l = url\_label(u)$. We also require $l = url\_label(u')$ to prevent a password leakage when the asynchronous continuation of an AJAX request is disclosed, as we anticipated in rule (I-XHRRESP). Moreover, FF^+ strips the cookies from outgoing requests when $url\_label(u) \neq url\_label(u')$ or $q = \times$: this prevents both classic and local CSRF attacks. Notice that only untainted pages are allowed to open untainted network connections via XHR.

We conclude with rule (O-LOGIN). The condition $\rho(c) = url\_label(u)$ prevents login CSRF attacks, where the user is authenticated as the attacker, while the requirement $l = url\_label(u)$ ensures the confidentiality of the password. We also require that any login form is submitted to a URL within the same origin of the page: this prevents the attacker from fooling the user into establishing new authenticated sessions with trusted websites, which would violate session integrity.

E. Formal results

We can prove that FF^+ enforces session integrity for any *well-formed* trace. Intuitively, well-formedness ensures a basic set of constraints on incoming input events, which are needed for our formal result, but have a limited practical impact. Clearly, we do not assume that the intruder is forced to produce well-formed inputs.

TABLE 3 Reactive semantics of FF^{+}: inputs

(I-LOAD)
$$\frac{ck = \mathit{get\_http\_ck}(K, u)}{\langle W, K, N, \{\}, [\,]\rangle \xrightarrow{\mathsf{load}(u)} \langle W, K, N \uplus \{n \mapsto (u, (), \checkmark)\}, \{\}, \mathsf{doc\_req}(ck, u)\rangle}$$

(I-TEXT)
$$\frac{W(p) = (u, h, h', q) \qquad h(k) = \lambda x.e}{\langle W, K, N, \{\}, [\,]\rangle \xrightarrow{\mathsf{text}(p,k,n)} \langle W, K, N, \{p \mapsto (e\{n/x\}, \rho(n))\}, [\,]\rangle}$$

(I-DOCRESP)
$$\frac{q = \checkmark \Rightarrow K' = \mathit{sec\_upd\_ck}(K, u, ck) \qquad q = \times \Rightarrow K' = K}{\langle W, K, N \uplus \{n \mapsto (u, (), q)\}, \{\}, [\,]\rangle \xrightarrow{\mathsf{doc\_resp}(n,ck,u,\mathsf{blank},h,e)} \langle W \uplus \{p \mapsto (u, h, \{\}, q)\}, K', N, \{p \mapsto (e, \bot)\}, [\,]\rangle}$$

(I-DOCREDIR)
$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = \mathit{sec\_upd\_ck}(K, u, ck) \qquad q = \times \Rightarrow K' = K \\ q = \checkmark \wedge \mathit{url\_label}(u) = \mathit{url\_label}(u') \Rightarrow ck' = \mathit{get\_http\_ck}(K', u') \wedge q' = \checkmark \\ q = \times \vee \mathit{url\_label}(u) \neq \mathit{url\_label}(u') \Rightarrow ck' = \{\} \wedge q' = \times \end{array}}{\langle W, K, N \uplus \{n \mapsto (u, (), q)\}, \{\}, [\,]\rangle \xrightarrow{\mathsf{doc\_resp}(n,ck,u,u',h,e)} \langle W, K', N \uplus \{n \mapsto (u', (), q')\}, \{\}, \mathsf{doc\_req}(ck', u')\rangle}$$

(I-XHRRESP)
$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = \mathit{sec\_upd\_ck}(K, u, ck) \qquad q = \times \Rightarrow K' = K \\ h' = h'' \uplus \{n \mapsto \lambda x.e\} \qquad W' = W \uplus \{p \mapsto (u', h, h'', q)\} \qquad l = \mathit{url\_label}(u') \end{array}}{\langle W \uplus \{p \mapsto (u', h, h', q)\}, K, N \uplus \{n \mapsto (u, p, q)\}, \{\}, [\,]\rangle \xrightarrow{\mathsf{xhr\_resp}(n,ck,u,\mathsf{blank},v)} \langle W', K', N, \{p \mapsto (e\{v/x\}, l)\}, [\,]\rangle}$$

(I-XHRREDIR)
$$\frac{\begin{array}{c} q = \checkmark \Rightarrow K' = \mathit{sec\_upd\_ck}(K, u, ck) \qquad q = \times \Rightarrow K' = K \\ q = \checkmark \wedge \mathit{url\_label}(u) = \mathit{url\_label}(u') \Rightarrow ck' = \mathit{get\_http\_ck}(K', u') \wedge q' = \checkmark \\ q = \times \vee \mathit{url\_label}(u) \neq \mathit{url\_label}(u') \Rightarrow ck' = \{\} \wedge q' = \times \end{array}}{\langle W, K, N \uplus \{n \mapsto (u, p, q)\}, \{\}, [\,]\rangle \xrightarrow{\mathsf{xhr\_resp}(n,ck,u,u',v)} \langle W, K', N \uplus \{n \mapsto (u', p, q')\}, \{\}, \mathsf{xhr\_req}(ck', u')\rangle}$$

(I-MIRROR)
$$\frac{C \overset{i}{\mapsto} P}{C \overset{i}{\rightarrow} P}$$

(I-COMPLETE)
$$\frac{\langle W, K, N, \{\}, [\,]\rangle \overset{i}{\not\mapsto}}{\langle W, K, N, \{\}, [\,]\rangle \overset{i}{\rightarrow} \langle W, K, N, \{\}, \bullet\rangle}$$

Notation: we write $C \overset{i}{\not\mapsto}$ whenever there does not exist P such that $C \overset{i}{\mapsto} P$.

We say that a URL u is well-formed (written $\vdash_\diamond u$) iff $\mathit{domain}(u) \in \mathcal{N}_\perp$ and there exists $l \sqsubseteq \mathit{url\_label}(u)$ such that $\mathit{path}(u) \in \mathcal{N}_l$.

Definition 6 (Well-formed Trace)**.** *An input event i is* well-formed *if and only if the judgement $\vdash_\diamond i$ can be proved through the following inference rules:*

(WF-LOAD)
$$\frac{\vdash_\diamond u}{\vdash_\diamond \mathsf{load}(u)}$$

(WF-TEXT)
$$\frac{n \in \mathcal{N}_{\rho(n)}}{\vdash_\diamond \mathsf{text}(p, k, n)}$$

(WF-XHR)
$$\frac{\begin{array}{c} l = \mathit{url\_label}(u) \qquad \mathit{ck\_vals}(ck) \subseteq \mathcal{N}_l \qquad \vdash_\diamond u \qquad \vdash_\diamond u' \\ \exists l' \sqsubseteq l : \mathit{path}(u') \in \mathcal{N}_{l'} \qquad \mathit{dom}(ck) \cup \mathit{fn}(v) \cup \{n\} \subseteq \mathcal{N}_\perp \end{array}}{\vdash_\diamond \mathsf{xhr\_resp}(n, ck, u, u', v)}$$

(WF-DOC)
$$\frac{\begin{array}{c} l = \mathit{url\_label}(u) \qquad \mathit{ck\_vals}(ck) \subseteq \mathcal{N}_l \\ \vdash_\diamond u \qquad \vdash_\diamond u' \qquad \exists l' \sqsubseteq l : \mathit{path}(u') \in \mathcal{N}_{l'} \\ \mathit{dom}(ck) \cup \mathit{fn}(h) \cup \mathit{fn}(e) \cup \{n\} \subseteq \mathcal{N}_\perp \end{array}}{\vdash_\diamond \mathsf{doc\_resp}(n, ck, u, u', h, e)}$$

We say that a trace (I, O) is well-formed iff so is every $i \in I$.

An explanation of the rules follows: rule (WF-LOAD) ensures that the user never types in the address bar a URL containing a password (or an authentication cookie value) which should not be disclosed to the remote server. Rule (WF-TEXT) rules out text inputs containing names corresponding to authentication cookie values: in

TABLE 4 Reactive semantics of FF^{+}: outputs

(O-APP)
$$\frac{}{\langle W, K, N, \{p \mapsto ((\lambda x.e)\ v, l)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto (e\{v/x\}, l)\}, [\,]\rangle}$$

(O-LETCTX)
$$\frac{\langle W, K, N, \{p \mapsto (e', l)\}, [\,]\rangle \overset{o}{\mapsto} \langle W', K', N', \{p \mapsto (e'', l)\}, [\,]\rangle}{\langle W, K, N, \{p \mapsto (\mathsf{let}\ x = e'\ \mathsf{in}\ e, l)\}, [\,]\rangle \overset{o}{\mapsto} \langle W', K', N', \{p \mapsto (\mathsf{let}\ x = e''\ \mathsf{in}\ e, l)\}, [\,]\rangle}$$

(O-LET)
$$\frac{}{\langle W, K, N, \{p \mapsto (\mathsf{let}\ x = v\ \mathsf{in}\ e, l)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto (e\{v/x\}, l)\}, [\,]\rangle}$$

(O-GET)
$$\frac{W(p) = (u, h, h', q) \qquad d = \mathit{domain}(u) \qquad \exists ck : K(d) = ck \wedge ck(k) = (n, f) \wedge f \in \{\bot, \mathtt{S}\}}{\langle W, K, N, \{p \mapsto (k?, l)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto (n, l)\}, [\,]\rangle}$$

(O-GETFAIL)
$$\frac{W(p) = (u, h, h', q) \qquad d = \mathit{domain}(u) \qquad \neg\exists ck : K(d) = ck \wedge ck(k) = (n, f) \wedge f \in \{\bot, \mathtt{S}\}}{\langle W, K, N, \{p \mapsto (k?, l)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto ((), l)\}, [\,]\rangle}$$

(O-SET)
$$\frac{\begin{array}{c} W(p) = (u, h, h', q) \qquad d = \mathit{domain}(u) \\ \neg\exists ck : K(d) = ck \wedge ck(k) = (m, f') \wedge f' \in \{\mathtt{H}, \top\} \qquad K' = \mathit{upd\_ck}(K, d, \{k \mapsto (n, f)\}) \end{array}}{\langle W, K, N, \{p \mapsto (k!\langle n, f\rangle, \bot)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K', N, \{p \mapsto ((), \bot)\}, [\,]\rangle}$$

(O-SETFAIL)
$$\frac{W(p) = (u, h, h', q) \qquad d = \mathit{domain}(u) \qquad l \neq \bot \vee (\exists ck : K(d) = ck \wedge ck(k) = (m, f') \wedge f' \in \{\mathtt{H}, \top\})}{\langle W, K, N, \{p \mapsto (k!\langle n, f\rangle, l)\}, [\,]\rangle \overset{\bullet}{\mapsto} \langle W, K, N, \{p \mapsto ((), l)\}, [\,]\rangle}$$

(O-XHR)
$$\frac{\begin{array}{c} W' = W \uplus \{p \mapsto (u', h, h' \uplus \{n \mapsto \lambda x.e\}, q)\} \\ l \neq \bot \Rightarrow l = \mathit{url\_label}(u) = \mathit{url\_label}(u') \wedge q = \checkmark \\ q = \checkmark \wedge \mathit{url\_label}(u) = \mathit{url\_label}(u') \Rightarrow ck = \mathit{get\_http\_ck}(K, u) \wedge q' = \checkmark \\ q = \times \vee \mathit{url\_label}(u) \neq \mathit{url\_label}(u') \Rightarrow ck = \{\} \wedge q' = \times \end{array}}{\langle W \uplus \{p \mapsto (u', h, h', q)\}, K, N, \{p \mapsto (\mathsf{xhr}(u, \lambda x.e), l)\}, [\,]\rangle \xmapsto{\mathsf{xhr\_req}(ck, u)} \langle W', K, N \uplus \{n \mapsto (u, p, q')\}, \{p \mapsto ((), l)\}, [\,]\rangle}$$

(O-LOGIN)
$$\frac{W(p) = (u', h, h', \checkmark) \qquad \rho(c) = \mathit{url\_label}(u) \qquad l = \mathit{url\_label}(u) = \mathit{url\_label}(u') \qquad ck = \mathit{get\_http\_ck}(K, u)}{\langle W, K, N, \{p \mapsto (\mathsf{auth}(u, c), l)\}, [\,]\rangle \xmapsto{\mathsf{login}(ck, u, c)} \langle W, K, N \uplus \{n \mapsto (u, (), \checkmark)\}, \{p \mapsto ((), l)\}, [\,]\rangle}$$

(O-FLUSH)
$$\frac{}{\langle W, K, N, T, o\rangle \overset{o}{\mapsto} \langle W, K, N, T, [\,]\rangle}$$

(O-MIRROR)
$$\frac{P \overset{o}{\mapsto} Q}{P \overset{o}{\to} Q}$$

(O-COMPLETE)
$$\frac{\langle W, K, N, \{p \mapsto (e, l)\}, [\,]\rangle \not\mapsto}{\langle W, K, N, \{p \mapsto (e, l)\}, [\,]\rangle \overset{\bullet}{\to} \langle W, K, N, \{\}, [\,]\rangle}$$

Notation: we write $P \not\mapsto$ whenever there do not exist o and Q such that $P \overset{o}{\mapsto} Q$.

other words, we assume that the user is always entering either a password or some public data. Rules (WF-DOC) and (WF-XHR) ensure that cookies set by a honest server are picked from the correct name partition and only occur in the standard HTTP header: furthermore, we require that confidential data (e.g., passwords) never appear in the body of a response or in the cookie names.

Theorem 1 (Session Integrity)**.** FF^+ *enforces session integrity for any well-formed trace (with respect to the threat model in Section III-C).*

The proof draws on a label-indexed family of *simulation* relations, which connect the attacked trace with the original one. The proof is challenging, due to the significant differences which may arise between the two traces: full details are in [2].

IV. Enforcing Session Integrity in SessInt

In this section, we discuss how to transfer FF^+ provable security into real browser security. To accomplish that, the following aspects must be taken in due account. First, the implementation of the required protection mechanisms should be designed so as to minimize their impact on the user experience: this is a difficult task, which requires careful design based on the search of the best possible trade-off between security and usability. Second, the design should lend itself to an implementation as a browser extension, to ease its deployment. When that is not possible, it is important to fine-tune the proposed security mechanisms so that they are still consistent with the theoretical model.

Below, we report on our development of SessInt, a proof-of-concept implementation of the integrity mechanisms of FF^+ as a browser extension for Google Chrome. While the current design is targeted at Chrome, and depends on its extension API, the same development appears possible on other major web browsers.

A. Implementing FF^+ security

We start by discussing how the different browsing events correspond to FF^+ events, and are handled by SessInt accordingly.

1) Address bar: Typing a URL in the address bar and loading a page corresponds to the load event in FF^+, i.e., we trust what the user types in the address bar. Here we have a first problem due to Chrome's API, which does not provide enough information to distinguish between a request triggered by the user typing in the address bar and a redirect, possibly caused by Javascript. For the moment, we have solved the issue by letting the user add a special character 'g' before inserting the URL, so that we can capture this input via the Chrome omnibox API and detect, accordingly, that the URL has been typed in the address bar.

2) User clicks: Following a link via a click is mapped to a xhr operation of FF^+. The rationale is that we do not trust clicks as, in fact, they might have been performed by malicious Javascript code. Moreover, it is unrealistic to assume that the user carefully checks every single link before clicking on it. Even though a user click could correspond to a load event, we decided to treat it as an xhr_req event and apply a more conservative security policy, whereby SessInt strips all authentication cookies before sending cross-origin requests, so as to prevent cross-origin request forgeries from malicious scripts.

3) Implicit loads: Implicit loads from a page or script correspond to xhr operations in FF^+, since we cannot trust these events. Hence, SessInt strips all authentication cookies before sending cross-origin requests.

4) Passwords: In order to prevent passwords from being leaked by malicious Javascript code, we sandbox login forms into an isolated popup. SessInt implements a password manager, which checks that the input password is correct before sending it. If the password is not yet in the password manager, the user is asked for confirmation and, in case of a positive answer, the password is stored and associated to the page and action URLs, to enforce the runtime discipline adopted by FF^+. Notice that the Chrome API does not allow to inspect the page content before inline scripts are executed. However, if these scripts modify the action URL before the extension creates the sandboxed form, the password manager will detect it by a comparison with the stored action URL and will warn the user before proceeding.

5) Cookies: SessInt performs a taint tracking over the open network connections, exactly as FF^+: cookies are only updated when they are received over an untainted connection. SessInt marks any authentication cookie received by the browser on HTTP connections as `HttpOnly`; authentication cookies received over HTTPS, instead, are marked as both `HttpOnly` and `Secure`. This prevents leakage from malicious Javascript programs and protects cookies in case HTTP links are injected into HTTPS websites. To preserve functionality, SessInt forces a redirection on HTTPS for the entire website when a login form is submitted over HTTPS: indeed, if the website contains some hard-coded HTTP links, marking some authentication cookies as `Secure` would break the session when navigating these links. As done by other extensions [15], [20], [33], authentication cookies are detected based on standard naming conventions (e.g., `PHPSESSID`) and a heuristics that measures the degree of entropy of the cookie value. In a recent paper we show how the authentication cookie detection process

can be improved significantly using machine learning techniques [16].

B. Protection vs usability

There are a few situations where the security policy of FF^+ would break too many websites, hence we have to slightly relax it in SESSINT. We discuss these situations and the implications on browser security and usability.

1) HTTP sessions: Some websites only support HTTPS for a subset of their pages. If some portions of the website do not provide support for HTTPS, SESSINT selectively allows a fallback to HTTP, with the proviso that cookies which have been previously promoted to `Secure` by the extension must be included to preserve the session [15]. If a HTTPS connection times out, we do not force the fallback, to prevent a network attacker intercepting HTTPS traffic from forcing SESSINT into leaking over HTTP authentication cookies of websites normally providing HTTPS support. Of course, we cannot provide session security against network attacks for websites which only partially support HTTPS, and the user is warned when this is the case.

2) Redirection to HTTPS: Many websites redirect the browser to HTTPS when an HTTP access is requested. However, SESSINT would not include authentication cookies upon these HTTPS redirections, since these redirects could as well be exploited by a network attacker to point the browser to a sensitive HTTPS URL and carry out a forgery: this cookie stripping breaks many websites, e.g., Facebook. To regain functionality, in the specific case of a protocol upgrading *with unmodified URL*, the user is asked (once for each site) to confirm that the redirection is expected, so that authentication cookies can be sent to the website. If the redirection looks suspicious, the user can block it.

3) HTTPS login forms into HTTP pages: It is common to find websites where HTTPS login forms are embedded (e.g., as iframes) into HTTP pages. This is insecure, as an attacker can change the HTTP page so as to redirect forms to a server that he controls, but it is a very common practice and we need to let it work. Our choice is to warn the user when this happens, then the password manager will give an extra warning in case the password is going to be sent to a URL that is not yet known. The combination of the two warnings should make the user well aware of a possible attack.

4) Subdomains and external sites: It is common that secure sessions link to subdomains or to external sites as, e.g., in e-payments. Navigating to a different domain would normally strip authentication cookies. To avoid breaking websites, SESSINT by default sends authentication cookies when moving into a subdomain, even though this could sometimes be exploited by a web attacker with scripting capabilities in the subdomain [14], [20]. We are investigating how to extend this behaviour to external (trusted) websites. A simple idea might be to include a white-list of trusted sites, e.g., for e-payments, that are needed to be reached by other websites, so that when the navigation comes back to the original site, authentication cookies are correctly sent and the session is preserved. We also plan to study to which extent we can engineer in SESSINT previous proposals aimed at supporting useful collaborative web scenarios [19].

C. Experiments

We tested SESSINT on existing vulnerable web applications, such as OWASP Mutillidae and Damn Vulnerable Web Application. We believe this is important to confirm that the more relaxed security policy adopted in SESSINT does not sacrifice too much of the bullet-proof security of FF^+. Here, we report some simple, but significant examples of attacks prevented by SESSINT:

1) CSRF: A link to domain A from a different domain B, that performs an action inside an active session with A (cf. Figure 1 (a)). With SESSINT authentication cookies are stripped and the action has no effect.

2) Cookie stealing via XSS: An XSS attack can access the JavaScript object `document.cookie` and leak an authentication cookie (cf. Figure 1 (b)). With SESSINT all authentication cookies are set `HttpOnly` and the attack is prevented.

3) Local CSRF: Domain A is vulnerable to XSS. The attacker injects a payload into A that redirects to a location, still in A, that performs an action inside the session (cf. Figure 1 (c)). The attack is prevented by SESSINT, since authentication cookies are stripped when a redirection happens over a tainted connection.

V. CONCLUSION

We introduced a novel notion of web session integrity and we showed that our definition is both general and amenable for client-side enforcement. We then proposed FF^+, a security-enhanced model of a web browser that provides a full-fledged and provably sound enforcement of web session integrity. Based on that, we developed SESSINT, a proof-of-concept browser extension which implements the security checks formalized in FF^+. We discussed the effectiveness of our solution and we presented some design choices we made to foster usability.

As a future work, we would like to further engineer SESSINT, trying to support more complicated collaborative web scenarios. We also plan to investigate how to enforce web session integrity in a browser supporting information flow control policies like FlowFox [24].

Acknowledgements: We would like to thank the anonymous referees for their valuable comments and our shepherd Tamara Rezk for her guidance in improving the original submission of this paper. This work was partially supported by the following MIUR Projects: PON ADAPT, PRIN CINA and PRIN Security Horizons.

REFERENCES

[1] HTTP state management mechanism. http://tools.ietf.org/html/rfc6265.

[2] Provably Sound Browser-Based Enforcement of Web Session Integrity (full version). http://www.dais.unive.it/~calzavara/csf14-full.pdf.

[3] The Transport Layer Security (TLS) protocol. http://tools.ietf.org/html/rfc4346.

[4] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *International Conference on the World Wide Web (WWW)*, pages 517–524, 2008.

[5] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 290–304, 2010.

[6] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based single sign-on protocols: Impact and remediations. *Computers & Security*, 33:41–58, 2013.

[7] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Formal Methods in Security Engineering (FMSE)*, pages 1–10, 2008.

[8] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *Principles of Security and Trust (POST)*, pages 126–146, 2013.

[9] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 247–262, 2012.

[10] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security (CCS)*, pages 75–88, 2008.

[11] A. Bohannon. *Foundations of webscript security*. PhD thesis, University of Pennsylvania, 2012.

[12] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)*, pages 1–12, 2010.

[13] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security (CCS)*, pages 79–90, 2009.

[14] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for web applications. In *Web 2.0 Security & Privacy (W2SP)*, 2011.

[15] 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 (ESSoS)*, pages 161–178, 2014.

[16] S. Calzavara, G. Tolomei, M. Bugliesi, and S. Orlando. Quite a mess in my cookie jar! Leveraging machine learning to protect web authentication. In *International Conference on World Wide Web (WWW)*, pages 189–200, 2014.

[17] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology*, 12(1):1, 2012.

[18] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems (ESSoS)*, pages 18–34, 2010.

[19] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *European Symposium on Research in Computer Security (ESORICS)*, pages 100–116, 2011.

[20] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: Self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 59–72, 2012.

[21] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[22] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *Principles of Programming Languages (POPL)*, pages 323–335, 2008.

[23] B. S. Y. Fung and P. P. C. Lee. A privacy-preserving defense mechanism against request forgery attacks. In *International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, pages 45–52, 2011.

[24] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security (CCS)*, pages 748–759, 2012.

[25] P. A. Hallgren, D. T. Mauritzson, and A. Sabelfeld. Glasstube: A lightweight approach to web application integrity. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 71–82, 2013.

[26] C. Jackson and A. Barth. ForceHTTPS: protecting high-security web sites from network attacks. In *International Conference on World Wide Web (WWW)*, pages 525–534, 2008.

[27] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable protection against session fixation attacks. In *ACM Symposium on Applied Computing (SAC)*, pages 1531–1537, 2011.

[28] M. Johns, S. Lekies, B. Braun, and B. Flesch. BetterAuth: web authentication revisited. In *Annual Computer Security Applications Conference (ACSAC)*, pages 169–178, 2012.

[29] M. Johns and J. Winter. RequestRodeo: client side protection against session riding. *Proceedings of the OWASP Europe Conference*, pages 5–17, 2006.

[30] E. Kirda, C. Krügel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM Symposium on Applied Computing (SAC)*, pages 330–337, 2006.

[31] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography (FC)*, pages 238–255, 2009.

[32] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[33] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight protection against session hijacking. In *Engineering Secure Software and Systems (ESSoS)*, pages 87–100, 2011.

[34] N. Nikiforakis, Y. Younan, and W. Joosen. Hproxy: Client-side detection of ssl stripping attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 200–218, 2010.

[35] S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *ACM Conference on Computer and Communications Security (CCS)*, pages 615–626, 2011.

[36] Y. Zhou and D. Evans. Why aren't HTTP-Only cookies more widely deployed. In *Web 2.0 Security and Privacy Workshop (W2SP'10)*, 2010.