Language-Based Web Session Integrity

Stefano Calzavara*, Riccardo Focardi*, Niklas Grimm†, Matteo Maffei†, Mauro Tempesta†
*Università Ca' Foscari Venezia †TU Wien

***Abstract*—Session management is a fundamental component of web applications: despite the apparent simplicity, correctly implementing web sessions is extremely tricky, as witnessed by the large number of existing attacks. This motivated the design of formal methods to rigorously reason about web session security which, however, are not supported at present by suitable automated verification techniques. In this paper we introduce the first security type system that enforces session security on a core model of web applications, focusing in particular on server-side code. We showcase the expressiveness of our type system by analyzing the session management logic of HotCRP, Moodle, and phpMyAdmin, unveiling novel security flaws that have been acknowledged by software developers.**

I. Introduction

Since the HTTP protocol is stateless, web applications that need to keep track of state information over multiple HTTP requests have to implement custom logic for *session management*. Web sessions typically start with the submission of a login form from a web browser, where a registered user provides her access credentials to the web application. If these credentials are valid, the web application stores in the user's browser fresh *session cookies*, which are automatically attached to all subsequent requests sent to the web application. These cookies contain enough information to authenticate the user and to keep track of session state across requests.

Session management is essential in the modern Web, yet it is often vulnerable to a range of attacks and surprisingly hard to get right. For instance, the theft of session cookies allows an attacker to impersonate the victim at the web application [31], [12], [33], while the weak integrity guarantees offered by cookies allow subtle attacks like cookie forcing, where a user is forced into an attacker-controlled session via cookie overwriting [36]. Other common attacks include cross-site request forgery (CSRF) [28], where an attacker instruments the victim's browser to send forged authenticated requests to a target web application, and login CSRF, where the victim's browser is forced into the attacker's session by submitting a login form with the attacker's credentials [9]. We refer to a recent survey for an overview of attacks against web sessions and countermeasures [16].

Given the complexity of session management and the range of threats to be faced on the web, a formal understanding of web session security and the design of automated verification techniques is an important research direction. Web sessions and their desired security properties have been formally studied in several papers developing browser-side defenses for web sessions [13], [12], [29], [14]: while the focus on browser-side protection mechanisms is appealing to protect users of vulnerable web applications, the deployment of these solutions is limited since it is hard to design browser-side defenses that do not cause compatibility issues on existing websites and are effective enough to be integrated in commercial browsers [16].

Thus, security-conscious developers would better rely on server-side programming practices to enforce web session security when web applications are accessed by standard browsers. Recently, Fett et al. [22] formalized a session integrity property specific to OpenID within the Web Infrastructure Model (WIM), an expressive web model within which proofs are, however, manual and require a strong expertise.

In this work, we present *the first static analysis technique for web session integrity*, focusing on sound server-side programming practices. In particular:

1) we introduce a core formal model of web systems, representing browsers, servers, and attackers who may mediate communications between them. Attackers can also interact with honest servers to establish their own sessions and host malicious content on compromised websites. The goal in the design of the model is to retain simplicity, to ease the presentation of the basic principles underlying our analysis technique, while being expressive enough to capture the salient aspects of session management in real-world case studies. In this model, we formalize a generic definition of *session integrity*, inspired by prior work on browser-side security [13], as a semantic hyperproperty [18] ruling out a wide range of attacks against web sessions;
2) we design a novel type system for the verification of session integrity within our model. The type system exploits confidentiality and integrity guarantees of session data to endorse untrusted requests coming from the network and enforces appropriate browser-side invariants in the corresponding responses to guarantee session integrity;
3) we showcase the effectiveness and generality of our type system by analyzing the session management logic of HotCRP, Moodle, and phpMyAdmin. After encoding the relevant code fragments in our formal model, we use the type system to establish a session integrity proof: failures in this process led to the discovery of critical security flaws. We identified two vulnerabilities in HotCRP that allow an attacker to hijack accounts of authors and even reviewers, and one in phpMyAdmin, which has been assigned a CVE [30]. All vulnerabilities have been reported and acknowledged by the application developers. We finally established security proofs for the fixed versions by typing.

II. Overview

In this Section we provide a high-level overview of our approach to the verification of session integrity. Full formal details and a complete security analysis of the HotCRP conference management system are presented in the remainder of the paper.

A. Encoding PHP Code in our Calculus

The first step of our approach consists in accessing the PHP implementation of HotCRP and carefully handcrafting a model of its authentication management mechanisms into the core calculus we use to model web application code. While several commands are standard, our language for server-side programs includes some high-level commands abstracting functionalities that are implemented in several lines of PHP code. The **login** command abstracts a snippet of code checking, e.g., in a database, whether the provided credentials match an existing user in the system. Command **auth** is a *security assertion* parametrized by expressions it depends on. In our encoding it abstracts code performing security-sensitive operations within the active session: here it models code handling paper submissions in HotCRP. Command **start** takes as argument a session identifier and corresponds to the `session_start` function of PHP, restoring variables set in the session memory during previous requests bound to that session.

In the following we distinguish standard PHP variables from those stored in the session memory (i.e., variables in the `$_SESSION` array) using symbols @ and $, respectively. The **reply** command models the server's response in a structured way by separating the page's DOM, scripts, and cookies set via HTTP headers.

B. A Core Model of HotCRP

We assume that the HotCRP installation is hosted at the domain d_C and accessible via two HTTPS endpoints: *login*, where users perform authentication using their access credentials, and *manage*, where users can upload their papers or withdraw their submissions. The session management logic is based on a cookie sid established upon login. We now discuss the functionality of the two HTTPS endpoints; we denote the names of cookies in square brackets and the name of parameters in parentheses. The login endpoint expects a username uid and a password pwd used for authentication:

```
1. $login[](uid, pwd) \hookrightarrow$
2.   $\textbf{if } uid = \bot \textbf{ and } pwd = \bot \textbf{ then}$
3.     $\textbf{reply } (\{\texttt{auth} \mapsto \mathsf{form}(login, \langle\bot,\bot\rangle)\}, \textbf{skip}, \{\})$
4.   $\textbf{else}$
5.     $@r := fresh();\ \textbf{login } uid, pwd, @r;\ \textbf{start } @r;\ \$user := uid;$
6.     $\textbf{reply } (\{\texttt{link} \mapsto \mathsf{form}(manage, \langle\bot,\bot,\bot\rangle)\}, \textbf{skip}, \{sid \mapsto x\})$
7.       $\textbf{with } x = @r$
```

If the user contacts the endpoint without providing access credentials, the endpoint replies with a page containing a login form expecting the username and password (lines 2–3). Otherwise, upon successful authentication via uid and pwd, the endpoint starts a new session indexed by a fresh identifier which is stored into the variable $@r$ (line 5). For technical convenience, in the **login** command we also specify the fresh session identifier as a third parameter to bind the session with the identity of its owner. Next, the endpoint stores the user's identity in the session variable $\$user$ so that the session identifier can be used to authenticate the user in subsequent requests (line 5). Finally, the endpoint sends a reply to the user's browser which includes a link to the submission management interface and sets a cookie sid containing the session identifier stored in $@r$ (lines 6–7).

The submission management endpoint requires authentication, hence it expects a session cookie sid. It also expects three parameters: a $paper$, an $action$ (submit or withdraw) and a $token$ to protect against CSRF attacks [9]:

```
1. $manage[sid](paper, action, token) \hookrightarrow$
2.   $\textbf{start } @sid;$
3.   $\textbf{if } \$user = \bot \textbf{ then}$
4.     $\textbf{reply } (\{\texttt{auth} \mapsto \mathsf{form}(login, \langle\bot,\bot\rangle)\}, \textbf{skip}, \{\})$
5.   $\textbf{else if } paper = \bot \textbf{ then}$
6.     $\$utoken = fresh();$
7.     $\textbf{reply } (\{\texttt{add} \mapsto \mathsf{form}(manage, \langle\bot, \texttt{submit}, x\rangle),$
8.             $\texttt{del} \mapsto \mathsf{form}(manage, \langle\bot, \texttt{withdraw}, x\rangle)\}, \textbf{skip}, \{\})$
9.       $\textbf{with } x = \$utoken$
10.  $\textbf{else if tokenchk}(token, \$utoken) \textbf{ then}$
11.    $\textbf{auth } paper, action \textbf{ at } \ell_C;\ \textbf{reply } (\{\}, \textbf{skip}, \{\})$
```

The endpoint first tries to start a session over the cookie sid: if it identifies a valid session, session variables from previous requests are restored (line 2). The condition $\$user = \bot$ checks whether the session is authenticated, since the variable is only set after login: if it is not the case, the endpoint replies with a link to the login page (lines 3–4). If the user is authenticated but does not provide any paper in her request, the endpoint replies with two forms used to submit or withdraw a paper respectively. Such forms are protected against CSRF with a fresh token, whose value is stored in the session variable $\$utoken$ (lines 5–9). If the user is authenticated and requests an action over a given paper, the endpoint checks that the token supplied in the request matches the one stored in the user's session (line 10) and performs the requested action upon success (line 11). This is modeled via a security assertion in the code that authorizes the requested action on the paper on behalf of the owner of the session. The assertion has a security label ℓ_C, intuitively meaning that authorization can be trusted unless the attacker can read or write at ℓ_C. Security labels have a confidentiality and an integrity component, expressing who can read and who can write. They are typically used in the information flow literature [14] not only to represent the security of program terms but also the attacker itself. Here we let $\ell_C = (\mathsf{https}(d_C), \mathsf{https}(d_C))$, meaning that authorization can be trusted unless HTTPS communication with the domain d_C hosting HotCRP is compromised by the attacker.

C. Session Integrity

In this work, we are interested in *session integrity*. Inspired by [13], we formalize it as a relational property, comparing two different scenarios: an ideal world where the attacker does nothing and an attacked world where the attacker uses her capabilities to compromise the session. Intuitively, session integrity requires that any authorized action occurring in the attacked world can also happen in the ideal world, unless the attacker is powerful enough to void the security assertions; this must hold for all sequences of actions of a user interacting with the session using a standard web browser.

As a counterexample to session integrity for our HotCRP model, pick an attacker hosting an HTTPS website at the domain $d_E \neq d_C$, modeled by the security label $\ell_E = (\mathsf{https}(d_E), \mathsf{https}(d_E))$. Since $\ell_E \not\sqsupseteq \ell_C$, this attacker should not be able to interfere with authorized actions at the submission management endpoint. However, this does not hold due to the lack of CSRF protection on the endpoint *login*. In particular, pick the following sequence of user actions where *evil* stands for an HTTPS endpoint at d_E:

$$\begin{aligned}\vec{a} = {} & \mathsf{load}(1, \mathit{login}, \{\}),\ \mathsf{submit}(1, \mathit{login}, \mathtt{auth}, \{1 \mapsto \mathsf{usr}, 2 \mapsto \mathtt{pwd}\}), \\ & \mathsf{load}(2, \mathit{evil}, \{\}), \mathsf{submit}(1, \mathit{login}, \mathtt{link}, \{\}), \\ & \mathsf{submit}(1, \mathit{manage}, \mathtt{add}, \{1 \mapsto \mathtt{paper}\})\end{aligned}$$

The user opens the login endpoint in tab 1 and submits her username and password via the authentication form (identified by the tag `auth`). She then loads the attacker's website in tab 2 and moves back to tab 1 where she accesses the submission management endpoint by clicking the link obtained upon authentication. Finally, she submits a paper via the `add` form.

Session integrity is violated since the attacker can reply with a page containing a script which automatically submits the attacker's credentials to the login endpoint, authenticating the user as the attacker at HotCRP. Thus, the last user action triggers the security assertion in the attacker's session rather than in the user's session. Formally, this is captured by the security assertion firing the event $\sharp[\mathtt{paper}, \mathtt{submit}]^{\mathsf{usr},\mathsf{atk}}_{\ell_C}$, modeling that the paper is submitted by the user into the attacker's session. As such an event cannot be fired in the ideal world without the attacker, this violates session integrity.

In practice, an attacker could perform the attack against an author so that, upon uncareful submission, a paper is registered in the attacker's account, violating the paper's confidentiality. We also discovered a more severe attack allowing an attacker to log into the victim's session, explained in Section V.

D. Security by Typing

Our type system allows for sound verification of session integrity and is parametric with respect to an attacker label. In particular, typing ensures that the attacker has no way to forge authenticated events in the session of an honest user (as in a CSRF attack) or to force the user to perform actions within a session bound to the attacker's identity (e.g., due to a login CSRF). Failures arising during type-checking often highlight in a direct way session integrity flaws.

To ensure session integrity, we require two ingredients: first, we need to determine the identity of the sender of the request; second, we must ensure that the request is actually sent with the consent of the user, i.e., the browser is not sending the request as the attacker's deputy. Our type system captures these aspects using two labels: a *session* label and a *program counter* (PC) label. The session label models both the session's integrity (i.e., who can influence the session and its contents) and confidentiality (i.e., who can learn the session identifier used as access control token). Since the identity associated with an authenticated event is derived from the ongoing session, the session label captures the first ingredient. The PC label tracks who could have influenced the control flow to reach the current point of the execution. Since a CSRF attack is exactly a request of low integrity (as it is triggered by the attacker), this captures the second ingredient. Additionally, the type system relies on a typing environment that assigns types to URLs and their parameters, to local variables and to references in the server memory.

We type-check the code twice under different assumptions. First, we assume the scenario of an honest user regularly interacting with the page: here we assume that all URL parameters are typed according to the typing environment and we start with a high integrity PC label. Second, we assume the scenario of a CSRF attack where all URL parameters have low confidentiality and integrity (since they are controlled by the attacker) and we start with a low integrity PC label. In both cases, types for cookies and the server variables are taken from the typing environment since, even in a CSRF attack, cookies are taken from the cookie jar of the user's browser and the attacker has no direct access to the server memory.

We now explain on a high level why our type system fails to type-check our (vulnerable) HotCRP model. To type the security assertion $\mathbf{auth}\ \mathit{paper}, \mathit{action}\ \mathbf{at}\ \ell_C$ in the manage endpoint, we need a high integrity PC label, a high integrity session label and we require the parameters paper and action to be of high integrity. While the types of the parameters are immediately determined by the typing environment, the other two labels are influenced by the typing derivation.

In the CSRF scenario, the security assertion is unreachable due to the presence of the token check instruction (line 10). When typing, if we assume (in the typing environment) that $\$\mathit{ltoken}$ is a high confidentiality reference, we can conclude that the check always fails since the parameter token (controlled by the attacker) has low confidentiality, therefore we do not need to type-check the continuation.[1]

In the honest scenario, the PC label has high integrity assuming that all the preceding conditionals have high integrity guard expressions (lines 3 and 6). The session label is set in the command $\mathbf{start}\ @\mathit{sid}$ (line 2) and depends on the type of the session identifier $@\mathit{sid}$. To succeed in typing, $@\mathit{sid}$ must have high integrity. However, we cannot type-check the login endpoint under this assumption: since the code does not contain any command that allows pruning the CSRF typing branch (like the token check in the manage endpoint), the entire code must be typed with a low integrity PC label. This prevents typing the $\mathbf{reply}$ statement where cookie sid is set (lines 6–7), since writing to a high integrity location from a low integrity context is unsound. In practice, this failure in typing uncovers the vulnerability in our code: the integrity of the session cookie is low since an attacker can use a login CSRF attack to set a session cookie in the user's browser.

As a fix, one can protect the login endpoint against CSRF attempts by using *pre-sessions* [9]: when the login endpoint is visited for the first time by the browser, it creates a new

[1] This reasoning is sound only when credentials (e.g., session identifiers and CSRF tokens) are unguessable fresh names. To take into account these aspect, in the type system we have special types for references storing credentials (cf. Section IV-A) and we forbid subtyping for high confidentiality credentials.

unauthenticated session at the server-side (using a fresh cookie *pre*) and generates a token which is saved into the session and embedded into the login form. When submitting the login form, the contained token is compared to the one stored at the server-side in the pre-session and, if there is a mismatch, authentication fails:

```
1. $login[pre](uid, pwd, token) \hookrightarrow$
2.   if $uid = \bot$ and $pwd = \bot$ then
3.     $@r' := fresh()$; start $@r'$; $\$ltoken := fresh()$;
4.     reply $(\{\mathtt{auth} \mapsto \mathsf{form}(login, \langle \bot, \bot, x \rangle)\}, \mathbf{skip}, \{pre \mapsto y\})$
5.       with $x = \$ltoken, y = @r'$
6.   else
7.     start $@pre$;
8.     if tokenchk$(token, \$ltoken)$ then
9.       $@r := fresh()$; login $uid, pwd, @r$; start $@r$; $\$user := uid$;
10.      reply $(\{\mathtt{link} \mapsto \mathsf{form}(manage, \langle \bot, \bot, \bot \rangle)\}, \mathbf{skip}, \{sid \mapsto x\}$
11.        with $x = @r$
```

The session identified by *pre* has low integrity but high confidentiality: indeed, an attacker can cause a random *pre* cookie to be set in the user's browser (by forcing the browser to interact with the *login* endpoint), but she has no way to learn the value of the cookie and hence cannot access the session. We can thus assume high confidentiality for the session reference $\$ltoken$ in the session identified by *pre*.

With the proposed fix, the piece of code responsible for setting the session cookie *sid* is protected by a token check, where the parameter *token* is compared against the high confidentiality session reference $\$ltoken$ of the session identified by $@pre$ (line 8). Similar to the token check in the *manage* endpoint, this allows us to prune the CSRF typing branch and we can successfully type-check the code with a high integrity type for *sid*. We refer the reader to Section V-C for a detailed explanation of typing of the fixed *login* endpoint.

The HotCRP developer acknowledged the login CSRF vulnerability and the effectiveness of the proposed fix, which is currently under development.

III. A Formal Model of Web Systems

We present now our model of web systems that includes the relevant ingredients for modeling attacks against session integrity and the corresponding defenses and we formally define our session integrity property.

A. Expressiveness of the Model

Our model of browsers supports cookies and a minimal client-side scripting language featuring *i)* read/write access to the cookie jar and the DOM of pages; *ii)* the possibility to send network requests towards arbitrary endpoints and include their contents as scripts. The latter capability is used to model resource inclusion and a simplified way to perform XHR requests. In the model we can encode many security-sensitive aspects of cookies that are relevant for attacks involving their theft or overwriting, i.e., cookie prefixes [35] and attributes `Domain` and `Secure` [8]. We also model HSTS [27] which can improve the integrity guarantees of cookies set by HSTS-enabled domains. On the server-side we include primitives used for session management and standard defenses against CSRF attacks, e.g., double submit cookies, validation of the `Origin` header and the use of CSRF tokens.

For the sake of presentation and simplicity, we intentionally omit some web components that are instead covered in other web models (e.g., the WIM [22]) but are not fundamental for session integrity or for modelling our case studies. In particular, we do not model document frames and cross-frame communications via the Web Messaging API, web sockets, local storage, DNS and an equational theory for cryptographic primitives. We also exclude the `Referer` header since it conveys similar information to the `Origin` header which we already cover in our model. While we believe that our type system can be in principle extended to cover also these web components, the presentation and proof of soundness would become cumbersome, obfuscating the key aspects of our static analysis technique.

B. Syntax

We write $\vec{r} = \langle r_1, \ldots, r_m \rangle$ to denote a list of elements of length $m = |\vec{r}|$. We denote with r_k the k-th element of $\vec{r}$ and we let $r' :: \vec{r}$ be the list obtained by prepending the element r' to the list $\vec{r}$. A map M is a partial function from keys to values and we write $M(k) = v$ whenever the key k is bound to the value v in M. We let $dom(M)$ be the domain of M and $\{\}$ be the empty map. Given two maps M_1 and M_2, we define $M_1 \triangleleft M_2$ as the map M such that $M(k) = v$ iff either $M_2(k) = v$ or $k \notin dom(M_2)$ and $M_1(k) = v$. We write $M_1 \uplus M_2$ to denote $M_1 \triangleleft M_2$ if M_1 and M_2 are disjoint. We let $M\{k \mapsto v\}$ be the map obtained from M by substituting the value bound to k with v.

1) Basics: we let $\mathcal{N}$ be a set of names modeling secrets (e.g., passwords) and fresh identifiers that cannot be forged by an attacker. Names are annotated with a security label ℓ, that we omit in the semantics since it has no semantic effect. $\mathcal{R}$ is the set of references used to model cookies and memory locations, while $\mathcal{X}$ is the set of variables used for parameters and server commands. $\mathcal{I}$ is the set of identities representing users: we distinguish a special identity usr representing the honest user and we assume that the other identities are under the attacker's control.

A URL u is a triple (π, d, v) where $\pi \in \{\mathsf{http}, \mathsf{https}\}$ is the protocol identifier, d is the domain name, and v is a value encoding the path of the accessed resource. We ignore the port for the sake of simplicity. The origin of URL u is the simple label $\pi(d)$. For origins and URLs, we use $\bot$ for a blank value.

We let v range over values, i.e., names, primitive values (booleans, integers, etc.), URLs, identities and the blank value $\bot$. We use z to range both over values and variables.

A *page* is either the constant error or a map f representing the DOM of the page. The error page denotes that an error has occurred while processing a request at the server-side. The map f associates tags (i.e., strings) to links and HTML forms contained in the page. We represent them using the notation $\mathsf{form}(u, \vec{z})$, where u is the target URL and $\vec{z}$ is the list of parameters provided via the query string of a link or in the HTTP body of the request for forms.

TABLE I: Syntax (browsers B and scripts s are defined in the technical report [15]).

Basics

Names	$n^\ell, i^\ell, j^\ell \in \mathcal{N}$	References	$r \in \mathcal{R}$	Variables	$x \in \mathcal{X}$
Identities	$\iota \in \mathcal{I} \ni \mathsf{usr}$	Domains	$d \in \mathcal{D}$	URLs	$u \in \mathcal{U}$
Origins	$o \in \mathcal{O} \supseteq O$	Simple labels	$l \in \mathcal{L} \supseteq L$	Labels	$\ell ::= (l, l)$
Types	$\tau \in \mathcal{T}$	Numbers	$k, m \in \mathbb{N}$	Primitive values	$pv ::= true \mid false \mid k \mid \ldots$
Values	$v ::= pv \mid n \mid \iota \mid u \mid \bot \in \mathcal{V}$	Metavariables	$z \in \mathcal{V} \cup \mathcal{X}$	Forms	$f ::= \{\} \mid f \uplus \{v \mapsto \mathsf{form}(u, \vec{z})\}$
Pages	$page ::= \mathsf{error} \mid f$	Cookies	$ck ::= \{\} \mid ck \uplus \{r \mapsto z\}$	Memories	$M ::= \{\} \mid M \uplus \{r \mapsto v\}$

Servers

Expressions	$se ::= x \mid @r \mid \$r \mid v \mid fresh()^\ell \mid se \odot se'$	Commands	$c ::= \mathbf{skip} \mid \mathbf{halt} \mid c; c' \mid @r := se \mid \$r := se$
Environments	$E ::= i, \bot \mid i, j$		$\mid \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c' \mid \mathbf{login}\ se_u, se_{pw}, se_{id}$
Request contexts	$R ::= n, u, \iota, l$		$\mid \mathbf{start}\ se \mid \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell$
Databases	$D ::= \{\} \mid D \uplus \{n \mapsto M\}$		$\mid \mathbf{if\ tokenchk}(e, e')\ \mathbf{then}\ c$
Trust mappings	$\phi ::= \{\} \mid \phi \uplus \{n \mapsto \iota\}$		$\mid \mathbf{if\ originchk}(L)\ \mathbf{then}\ c$
Servers	$S ::= (D, \phi, t)$		$\mid \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$
Threads	$t ::= u[\vec{r}](\vec{x}) \hookrightarrow c \mid \lceil c \rfloor_E^R \mid t \parallel t$		$\mid \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$

User behavior

Tab IDs	$tab \in \mathbb{N}$
Inputs	$p ::= \{\} \mid p \uplus \{k \mapsto v^\tau\}$
Actions	$a ::= \mathsf{halt} \mid \mathsf{load}(tab, u, p) \mid \mathsf{submit}(tab, u, v, p)$

Web Systems

Attacker's Knowledge	$\mathcal{K} \subseteq \mathcal{N}$
Web Systems	$W ::= B \mid S \mid W \parallel W$
Attacked Systems	$A ::= (\ell, \mathcal{K}) \rhd W$

Memories are maps from references to values. We use them in the server to hold the values of the variables during the execution, while in the browser they are used to model the cookie jar. We stipulate that $M(r) = \bot$ if $r \notin dom(M)$, i.e., the access to a reference not in memory yields a blank value.

2) Server Model: we let se range over expressions including variables, references, values, sampling of a fresh name (with label ℓ), e.g., to generate fresh cookie values, and binary operations. Server-side applications are represented as commands featuring standard programming constructs and special instructions for session establishment and management. Command $\mathbf{login}\ se_u, se_{pw}, se_{id}$ models a login operation with username se_u and password se_{pw}. The identity of the user is bound to the session identifier obtained by evaluating se_{id}. Command $\mathbf{start}\ se$ starts a new session or restores a previous one identified by the value of the expression se. Command $\mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell$ produces an authenticated event that includes data identified by the list of expressions $\vec{se}$. The command is annotated with a label ℓ denoting the expected security level of the event which has a central role in the security definition presented in Section III-E. Commands $\mathbf{if\ tokenchk}(x, r)\ \mathbf{then}\ c$ and $\mathbf{if\ originchk}(L)\ \mathbf{then}\ c$ respectively model a token check, comparing the value of a parameter x against the value of the reference r, and an origin check, verifying whether the origin of the request occurs in the set L. These checks are used as a protection mechanism against CSRF attacks. Command $\mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$ outputs an HTTP response containing a $page$, a script s and a sequence of `Set-Cookie` headers represented by the map ck. This command is a binder for $\vec{x}$ with scope $page, s, ck$, that is, the occurrences of the variables $\vec{x}$ in $page, s, ck$ are substituted with the values obtained by evaluating the corresponding expressions in $\vec{se}$. Command $\mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x}$ outputs a message redirect to URL u with parameters $\vec{z}$ that sets the cookies in ck. This command is a binder for $\vec{x}$ with scope $\vec{z}, ck$.

Server code is evaluated using two memories: a global memory, freshly allocated when a connection is received, and a session memory, that is preserved across different requests. We write $@r$ and $\$r$ to denote the reference r in the global memory and in the session memory respectively. To link an executing command to its memories, we use an environment, which is a pair whose components identify the global memory and the session memory ($\bot$ when there is no active session).

The state of a server is modeled as a triple (D, ϕ, t) where the database D is a partial map from names to memories, ϕ maps session identifiers (i.e., names) to the corresponding user identities, and t is the parallel composition of multiple threads. Thread $u[\vec{r}](\vec{x}) \hookrightarrow c$ waits for an incoming connection to URL u and runs the command c when it is received. Lists $\vec{r}$ and $\vec{x}$ are respectively the list of cookies and parameters that the server expects to receive from the browser. Thread $\lceil c \rfloor_E^R$ denotes the execution of the command c in the environment E which identifies the memories of D on which the command operates. R tracks information about the request that triggered the execution, including the identifier n of the connection where the response by the server must be sent back, the URL of the endpoint u, the user ι who sent the request, and the origin of the request l. The user identity has no semantic import, but it is needed to spell out our security property.

3) User Behavior: action halt is used when an unexpected error occurs while browsing to prevent the user from performing further actions. Action $\mathsf{load}(tab, u, p)$ models the user entering the URL u in the address bar of her browser in tab, where p are the provided query parameters. Action $\mathsf{submit}(tab, u, v, p)$ models the user submitting a form or clicking on a link (identified by v) contained in the page at u rendered in tab; the parameters p are the inputs provided by the user. We represent user inputs as maps from integers to values v^τ annotated with their security type τ. In other words, we model that the user is aware of the security import of the provided parameters, e.g., whether a certain input is a password that must be kept confidential or a public value.

4) Browser Model: due to space constraints, we present the browser model in the technical report [15]. In the following we write $B_\iota(M, P, \vec{a})$ to represent a browser without any active script or open network connection, with cookie jar M and open pages P which is run by the user ι performing the list of actions $\vec{a}$.

5) Web Systems: the state of a web system is the parallel composition of the states of browsers and servers in the system. The state of an attacked web system also includes the attacker, modeled as a pair $(\ell, \mathcal{K})$ where the label ℓ defines the attacker power and $\mathcal{K}$ is her knowledge, i.e., a set of names that the attacker has learned by exploiting her capabilities.

C. Labels and Threat Model

Let $d \in \mathcal{D}$ be a domain and $\sim$ be the equivalence relation inducing the partition of $\mathcal{D}$ in sets of related domains.[2] We define the set of simple labels $\mathcal{L}$, ranged over by l, as the smallest set generated by the grammar:

$$l ::= \mathsf{http}(d) \mid \mathsf{https}(d) \mid l \vee l \mid l \wedge l$$

Intuitively, simple labels represent the entities entitled to read or write a certain piece of data, inspect or modify the messages exchanged over a network connection and characterize the capabilities of an attacker. A label ℓ is a pair of simple labels (l_C, l_I), where l_C and l_I are respectively the confidentiality and integrity components of ℓ. We let $C(\ell) = l_C$ and $I(\ell) = l_I$. We define the confidentiality pre-order $\sqsubseteq_C$ as the smallest pre-order on $\mathcal{L}$ closed under the following rules:

$$\frac{i \in \{1,2\}}{l_i \sqsubseteq_C l_1 \vee l_2} \qquad \frac{i \in \{1,2\}}{l_1 \wedge l_2 \sqsubseteq_C l_i} \qquad \frac{l_1 \sqsubseteq_C l_3 \quad l_2 \sqsubseteq_C l_3}{l_1 \vee l_2 \sqsubseteq_C l_3} \qquad \frac{l_1 \sqsubseteq_C l_2 \quad l_1 \sqsubseteq_C l_3}{l_1 \sqsubseteq_C l_2 \wedge l_3}$$

We define the integrity pre-order $\sqsubseteq_I$ on simple labels such that $\forall l, l' \in \mathcal{L}$ we have $l \sqsubseteq_I l'$ iff $l' \sqsubseteq_C l$, i.e., confidentiality and integrity are contra-variant. For $\sqsubseteq_C$ we define the operators $\sqcup_C$ and $\sqcap_C$ that respectively take the least upper bound and the greatest lower bound of two simple labels. We define analogous operators $\sqcup_I$ and $\sqcap_I$ for $\sqsubseteq_I$. We let $\ell \sqsubseteq \ell'$ iff $C(\ell) \sqsubseteq_C C(\ell') \wedge I(\ell) \sqsubseteq_I I(\ell')$. We also define bottom and top elements of the lattices as follows:

$$\bot_C = \textstyle\bigwedge_{d \in \mathcal{D}}(\mathsf{http}(d) \wedge \mathsf{https}(d)) \qquad \top_C = \textstyle\bigvee_{d \in \mathcal{D}}(\mathsf{http}(d) \vee \mathsf{https}(d))$$
$$\bot_I = \top_C \qquad \top_I = \bot_C \qquad \bot = (\bot_C, \bot_I) \qquad \top = (\top_C, \top_I)$$

We label URLs, user actions and cookies by means of the function λ. We label URLs with their origin, i.e., given $u = (\pi, d, v)$ we let $\lambda(u) = (\pi(d), \pi(d))$. The label is used to: 1) characterize the capabilities required by an attacker to read and modify the contents of messages exchanged over network connections towards u; 2) identify which cookies are sent to and can be set by u. The label of an action is the one of its URL, i.e., we let $\lambda(a) = \lambda(u)$ for $a = \mathsf{load}(tab, u, p)$ and $a = \mathsf{submit}(tab, u, v, p)$.

The labelling of cookies depend on several aspects, e.g., the attributes specified by the web developer. For instance, a cookie for the domain d is given the following label:

$$(\mathsf{http}(d) \wedge \mathsf{https}(d), \textstyle\bigwedge_{d' \sim d}(\mathsf{http}(d') \wedge \mathsf{https}(d')))$$

[2] Two domains are related if they share the same base domain, i.e., the first upper-level domain which is not included in the public suffix list [36]. For instance, `www.example.com` and `atk.example.com` are related domains, while `example.co.uk` and `atk.co.uk` are not.

The confidentiality label models that the cookie can be sent to d both over cleartext and encrypted connections, while the integrity component says that the cookie can be set by any of the related domains of d over any protocol, as dictated by the lax variant of the *Same Origin Policy* applied to cookies.

When the `Secure` attribute is used, the cookie is attached exclusively to HTTPS requests. However, `Secure` cookies can be set over HTTP [8], hence the integrity is unchanged. This behavior is represented by the following label:

$$(\mathsf{https}(d), \textstyle\bigwedge_{d' \sim d}(\mathsf{http}(d') \wedge \mathsf{https}(d')))$$

Cookie prefixes [35] are a novel proposal aimed at providing strong integrity guarantees for certain classes of cookies. In particular, compliant browsers ensure that cookies having names starting with the `__Secure-` prefix are set over HTTPS and the `Secure` attribute is set. In our label model they can be represented as follows:

$$(\mathsf{https}(d), \textstyle\bigwedge_{d' \sim d} \mathsf{https}(d'))$$

The `__Host-` prefix strengthens the policy enforced by `__Secure-` by additionally requiring that the `Domain` attribute is not set, thus preventing related domains from setting it. This is modeled by assigning the cookie the following label:

$$(\mathsf{https}(d), \mathsf{https}(d))$$

We provide additional examples, including the impact of HSTS on the labelling of cookies, in our technical report [15].

In the model we can also formalize popular attackers from the web security literature using labels which denote their read and write capabilities:

1) The web attacker hosts a malicious website on domain d. We assume that the attacker owns a valid certificate for d, thus the website is available both over HTTP and HTTPS:

$$(\mathsf{http}(d) \vee \mathsf{https}(d), \mathsf{http}(d) \vee \mathsf{https}(d))$$

2) The active network attacker can read and modify the contents of all HTTP communications:

$$(\textstyle\bigvee_{d \in \mathcal{D}} \mathsf{http}(d), \bigvee_{d \in \mathcal{D}} \mathsf{http}(d))$$

3) The related-domain attacker is a web attacker who hosts her website on a *related domain* of a domain d, thus she can set (domain) cookies for d. Assuming (for simplicity) that the attacker controls all the related domains of d, we can represent her capabilities with the following label:

$$(\textstyle\bigvee_{\substack{d' \sim d \\ d' \neq d}}(\mathsf{http}(d') \vee \mathsf{https}(d')), \; \bigvee_{\substack{d' \sim d \\ d' \neq d}}(\mathsf{http}(d') \vee \mathsf{https}(d')))$$

D. Semantics

We present now the most relevant rules of semantics in Table II, deferring to [15] for a complete formalization. In the rules we use the ternary operator "?:" with the usual meaning: $e \; ? \; e' : e''$ evaluates to e' if e is true, to e'' otherwise.

TABLE II: Semantics (excerpt).

Servers

(S-RECV)
$$\frac{\begin{array}{c}\alpha = \mathsf{req}(\iota_b, n, u, p, ck, l) \qquad R = n, u, \iota_b, l \qquad i \leftarrow \mathcal{N} \\ \forall k \in [1 \ldots |\vec{r}|].\, M(r_k) = (r_k \in \mathit{dom}(ck))\ ?\ ck(r_k) : \bot \qquad m = |\vec{x}| \\ \forall k \in [1 \ldots m].\, v_k = (k \in \mathit{dom}(p))\ ?\ p(k) : \bot \qquad \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m]\end{array}}{(D, \phi, u[\vec{r}](\vec{x}) \hookrightarrow c) \xrightarrow{\alpha} (D \uplus \{i \mapsto M\}, \phi, \lceil c\sigma \rfloor^{R}_{i,\bot} \parallel u[\vec{r}](\vec{x}) \hookrightarrow c)}$$

(S-RESTORESESSION)
$$\frac{E = i, \_ \qquad \mathit{eval}_E(se, D) = j \qquad j \in \mathit{dom}(D)}{(D, \phi, \lceil \mathbf{start}\ se \rfloor^{R}_{E}) \xrightarrow{\bullet} (D, \phi, \lceil \mathbf{skip} \rfloor^{R}_{i,j})}$$

(S-NEWSESSION)
$$\frac{E = i, \_ \qquad \mathit{eval}_E(se, D) = j \qquad j \notin \mathit{dom}(D)}{(D, \phi, \lceil \mathbf{start}\ se \rfloor^{R}_{E}) \xrightarrow{\bullet} (D \uplus \{j \mapsto \{\}\}, \phi, \lceil \mathbf{skip} \rfloor^{R}_{i,j})}$$

(S-LOGIN)
$$\frac{\begin{array}{c}R = n, u, \iota_b, l \qquad \mathit{eval}_E(se_u, D) = \iota_s \\ \mathit{eval}_E(se_{pw}, D) = \rho(\iota_s, u) \qquad \mathit{eval}_E(se_{id}, D) = j\end{array}}{(D, \phi, \lceil \mathbf{login}\ se_u, se_{pw}, se_{id} \rfloor^{R}_{E}) \xrightarrow{\bullet} (D, \phi \triangleleft \{j \mapsto \iota_s\}, \lceil \mathbf{skip} \rfloor^{R}_{E})}$$

(S-OCHKSUCC)
$$\frac{R = n, u, \iota_b, l \qquad l \in L}{(D, \phi, \lceil \mathbf{if\ originchk}(L)\ \mathbf{then}\ c \rfloor^{R}_{E}) \xrightarrow{\bullet} (D, \phi, \lceil c \rfloor^{R}_{E})}$$

(S-TCHKFAIL)
$$\frac{\mathit{eval}_E(e_1, D) \neq \mathit{eval}_E(e_2, D)}{(D, \phi, \lceil \mathbf{if\ tokenchk}(e_1, e_2)\ \mathbf{then}\ c \rfloor^{R}_{E}) \xrightarrow{\bullet} (D, \phi, \lceil \mathbf{reply}\ (\mathsf{error}, \mathbf{skip}, \{\}) \rfloor^{R}_{E})}$$

(S-AUTH)
$$\frac{\begin{array}{c}R = n, u, \iota_b, l \qquad j \in \mathit{dom}(\phi) \qquad \alpha = \sharp[\vec{v}]^{\iota_b, \phi(j)}_{\ell} \\ \forall k \in [1 \ldots |\vec{se}|].\, \mathit{eval}_{i,j}(se_k, D) = v_k\end{array}}{(D, \phi, \lceil \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell \rfloor^{R}_{i,j}) \xrightarrow{\alpha} (D, \phi, \lceil \mathbf{skip} \rfloor^{R}_{i,j})}$$

(S-REPLY)
$$\frac{\begin{array}{c}R = n, u, \iota_b, l \qquad m = |\vec{x}| = |\vec{se}| \qquad \forall k \in [1, m].\, \mathit{eval}_E(se_k, D) = v_k \\ \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m] \qquad \alpha = \overline{\mathsf{res}}(n, u, \bot, \langle\rangle, ck\sigma, page\sigma, s\sigma)\end{array}}{(D, \phi, \lceil \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} \rfloor^{R}_{E}) \xrightarrow{\alpha} (D, \phi, \lceil \mathbf{halt} \rfloor^{R}_{E})}$$

Web systems

(A-BROSER)
$$\frac{\begin{array}{c}W \xrightarrow{\overline{\mathsf{req}}(\iota_b, n, u, p, ck, l)} W' \qquad W' \xrightarrow{\mathsf{req}(\iota_b, n, u, p, ck, l)} W'' \\ \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell))\ ?\ (\mathcal{K} \cup ns(p, ck)) : \mathcal{K}\end{array}}{(\ell, \mathcal{K}) \triangleright W \xrightarrow{\bullet} (\ell, \mathcal{K}') \triangleright W''}$$

(A-BROATK)
$$\frac{\begin{array}{c}\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, l) \qquad W \xrightarrow{\alpha} W' \\ I(\ell) \sqsubseteq_I I(\lambda(u)) \\ \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell))\ ?\ (\mathcal{K} \cup ns(p, ck)) : \mathcal{K}\end{array}}{(\ell, \mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell, \mathcal{K}' \cup \{n\}) \triangleright W'}$$

(A-ATKSER)
$$\frac{\begin{array}{c}n \leftarrow \mathcal{N} \qquad \iota_b \neq \mathsf{usr} \qquad ns(p, ck) \subseteq \mathcal{K} \\ \alpha = \mathsf{req}(\iota_b, n, u, p, ck, l) \qquad W \xrightarrow{\alpha} W'\end{array}}{(\ell, \mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell, \mathcal{K} \cup \{n\}) \triangleright W'}$$

1) Servers: rules rely on the function $\mathit{eval}_E(se, D)$ that evaluates the expression se in the environment E using the database D. The formal definition is in [15], here we provide an intuitive explanation. The evaluation of $@r$ and $\$r$ yields the value associated to r in the global and the session memory identified by E, respectively. Expression $\mathit{fresh}()^{\ell}$ evaluates to a fresh name sampled from $\mathcal{N}$ with security label ℓ. A value evaluates to itself. Evaluation of binary operations is standard.

Rule (S-RECV) models the receiving of a connection n at the endpoint u, as indicated by the action $\mathsf{req}(\iota_b, n, u, p, ck, l)$. A new thread is spawned where command c is executed after substituting all the occurrences of variables in $\vec{x}$ with the parameters p received from the network. We use the value $\bot$ for uninitialized parameters. The environment is $i, \bot$ where i identifies a freshly allocated global memory and $\bot$ that there is no ongoing session. The references of the global memory in $\vec{r}$ are initialized with the values in ck (if provided). In the request context we include the details about the incoming connection, including the origin l of the page that produced the request (or $\bot$, e.g., when the user opens the page in a new tab). The thread keeps listening for other connections on the same endpoint.

The evaluation of command **start** se is modeled by rules (S-RESTORESESSION) and (S-NEWSESSION). If se evaluates to a name $j \in \mathit{dom}(D)$, we resume a previously established session, otherwise we create a new one and allocate a new empty memory that is added to the database D. We write $E = i, \_$ to denote that the second component of E is immaterial. In both cases the environment is updated accordingly.

Rule (S-LOGIN) models a successful login attempt. For this purpose, we presuppose the existence of a global partial function ρ mapping the pair (ι_s, u) to the correct password where ι_s is the identity of the user and u is the login endpoint. The rule updates the trust mapping ϕ by associating the session identifier specified in the **login** command with the identity ι_s.

Rules (S-OCHKSUCC) and (S-TCHKFAIL) treat a successful origin check and a failed token check, respectively. In the origin check we verify that the origin of the request is in a set of whitelisted origins, while in the token check we verify that two tokens match. In case of success we execute the continuation, otherwise we respond with an error message.

Rule (S-AUTH) produces the authenticated event $\sharp[\vec{v}]^{\iota_b, \iota_s}_{\ell}$ where $\vec{v}$ is data identifying the event, e.g., $paper$ and $action$ in the HotCRP example of Section II-B. The event is annotated with the identities ι_b, ι_s, representing the user running the browser and the account where the event occurred, and the label ℓ denoting the security level associated to the event.

Rule (S-REPLY) models a reply from the server over the open connection n as indicated by the action $\overline{\mathsf{res}}$. The response contains a page $page$, script s and a map of cookies ck, where all occurrences of variables in $\vec{x}$ are replaced with the evaluation results of the expressions in $\vec{se}$. The third and the fourth component of $\overline{\mathsf{res}}$ are the redirect URL and the corresponding parameters, hence we use $\bot$ to denote that no redirect happens. We stipulate that the execution terminates after performing the reply as denoted by the instruction **halt**.

2) Web Systems: the semantics of web systems regulates the communications among browsers, servers and the attacker. Rule (A-BROSER) synchronizes a browser sending a request $\overline{\mathsf{req}}$ with the server willing to process it, as denoted by the matching action req. Here the attacker does not play an active role (as denoted by action $\bullet$) but she may update her knowledge with new secrets if she can read the contents of the request, modeled by the condition $C(\lambda(u)) \sqsubseteq_C C(\ell)$.

Rule (A-BroAtk) uniformly models a communication from a browser to a server controlled by the attacker and an attacker that is actively intercepting network traffic sent by the browser. These cases are captured by the integrity check on the origin of the URL u. As in the previous rule, the attacker updates her knowledge if she can access the communication's contents. Additionally, she learns the network identifier needed to respond to the browser. In the trace of the system we expose the action intercepted/forged by the attacker. Rule (A-AtkSer) models an attacker opening a connection to an honest server. We require that the identity denoting the sender of the message belongs to the attacker and that the contents of the request can be produced by the attacker using her knowledge. Sequential application of the two rules lets us model a network attacker acting as a man-in-the-middle to modify the request sent by a browser to an honest server.

E. Security Definition

On a high level, our definition of session integrity requires that for each trace produced by the attacked web system, there exists a matching trace produced by the web system without the attacker, which in particular implies that authenticated actions cannot be modified or forged by the attacker. Before formalizing this property, we introduce the notion of trace.

Definition 1. *The system A generates the trace $\gamma = \alpha_1 \cdot \ldots \cdot \alpha_k$ iff the system can perform a sequence of steps $A \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} A'$ for some A' (also written as $A \xrightarrow{\gamma}^* A'$).*

Traces include attacker actions, authenticated events $\sharp[\vec{v}]_{\ell}^{\iota_b,\iota_s}$ and $\bullet$ denoting actions without visible effects or synchronizations not involving the attacker. Given a trace γ, we write $\gamma \downarrow (\iota, \ell)$ for the projection containing only the authentication events of the type $\sharp[\vec{v}]_{\ell}^{\iota_b,\iota_s}$ with $\iota \in \{\iota_b, \iota_s\}$. A trace γ is *unattacked* if it contains only $\bullet$ actions and authenticated events, otherwise γ is an *attacked* trace.

Now we introduce the definition of session integrity.

Definition 2. *A web system W preserves* session integrity *against the attacker $(\ell_a, \mathcal{K})$ for the honest user* usr *performing the actions $\vec{a}$ if for any attacked trace γ generated by the system $(\ell_a, \mathcal{K}) \triangleright B_{\mathsf{usr}}(\{\}, \{\}, \vec{a}) \parallel W$ there exists an unattacked trace γ' generated by the same system such that for all labels ℓ' we have:*

$$I(\ell_a) \not\sqsubseteq_I I(\ell') \Rightarrow \gamma \downarrow (\mathsf{usr}, \ell') = \gamma' \downarrow (\mathsf{usr}, \ell').$$

Intuitively, this means that the attacker can only produce authenticated events in her account or influence events produced by servers under her control. Apart from this, the attacker can only stop on-going sessions of the user but cannot intrude into them: this is captured by the existential quantification over unattacked traces that also lets us pick a prefix of any trace.

IV. Security Type System

We now present a security type system designed for the verification of session integrity on web applications. It consists of several typing judgments covering server programs and browser scripts. Due to space constraints, in this Section we cover only the part related to server-side code and refer to [15] for the typing rules of browser scripts.

A. Types

We introduce security types built upon the labels defined in Section III-C. We construct the set of security types $\mathcal{T}$, ranged over by τ, according to the following grammar:

$$\tau ::= \ell \mid \mathtt{cred}(\ell)$$

We also introduce the set of reference types $\mathcal{T}_{\mathcal{R}} = \{\mathtt{ref}(\tau) \mid \tau \in \mathcal{T}\}$ used for global and session references and we define the following projections on security types:

$$\mathit{label}(\ell) = \ell \quad \mathit{label}(\mathtt{cred}(\ell)) = \ell$$
$$I(\tau) = I(\mathit{label}(\tau)) \quad C(\tau) = C(\mathit{label}(\tau))$$

Security types extend the standard security lattice with the type $\mathtt{cred}(\ell)$ for credentials of label ℓ. We define the pre-order $\sqsubseteq_{\ell_a}$, parametrized by the attacker label ℓ_a, with the following rules:

$$\frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq_{\ell_a} \ell'} \qquad \frac{C(\tau) = C(\tau') \qquad I(\tau) \sqsubseteq_I I(\tau')}{\tau \sqsubseteq_{\ell_a} \tau'}$$

$$\frac{C(\tau) \sqcup_C C(\tau') \sqsubseteq_C C(\ell_a) \qquad I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau')}{\tau \sqsubseteq_{\ell_a} \tau'}$$

Intuitively, security types inherit the subtyping relation for labels but this is not lifted to the confidentiality label of credentials, since treating public values as secret credentials is unsound. However, types of low integrity and confidentiality (compared to the attacker's label) are always subtype of each other: in other words, we collapse all such types into a single one, as the attacker controls these values and is not limited by the restrictions enforced by types.

B. Typing Environment

Our typing environment $\Gamma = (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$ is a 5-tuple and conveys the following information:

- $\Gamma_{\mathcal{U}} : \mathcal{U} \to (\mathcal{L}^2 \times \vec{\mathcal{T}} \times \mathcal{L})$ maps URLs to labels capturing the security of the network connection, the types of the URL parameters and the integrity label of the reply;
- $\Gamma_{\mathcal{X}} : \mathcal{X} \to \mathcal{T}$ maps variables to types;
- $\Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$} : \mathcal{R} \to \mathcal{T}_{\mathcal{R}}$ map global references and session references, respectively, to reference types;
- $\Gamma_{\mathcal{V}} : \mathcal{V} \to (\mathcal{L}^2 \times \vec{\mathcal{T}} \times \mathcal{L})$ maps values used as tags for forms in the DOM to the corresponding type. We typically require the form's type to match the one of the form's target URL.

Now we introduce the notion of *well-formedness* which rules out inconsistent type assignments.

Definition 3. *A typing environment Γ is* well-formed *for λ and ℓ_a (written $\lambda, \ell_a, \Gamma \vdash \diamond$) if the following conditions hold:*

1) for all URLs $u \in \mathcal{U}$ with $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ we have:

a) $C(\ell_u) = C(\lambda(u)) \wedge I(\lambda(u)) \sqsubseteq_I I(\ell_u)$

b) for all $k \in [1 \ldots |\vec{\tau}|]$ we have $C(\tau_k) \sqsubseteq_C C(\ell_u) \wedge I(\ell_u) \sqsubseteq_I I(\tau_k)$

2) for all references $r \in \mathcal{R}$ with $\Gamma_{\mathcal{R}^@}(r) = \tau$:

a) $C(\tau) \sqsubseteq_C C(\lambda(r)) \wedge I(\lambda(r)) \sqsubseteq_I I(\tau)$

b) for all $u \in \mathcal{U}$, if $C(\lambda(r)) \sqsubseteq_C (\lambda(u)) \wedge I(\ell_a) \sqsubseteq_I I(\lambda(u))$ then $C(\tau) \sqsubseteq_C C(\ell_a)$

c) if $I(\ell_a) \sqsubseteq_I I(\lambda(r))$ and $\tau = \mathtt{cred}(\cdot)$ then $C(\tau) \sqsubseteq_C C(\ell_a)$

Conditions (1a) and (2a) ensure that the labels of URLs and cookies in the typing environment are at most as strict as in the function λ introduced in Section III-C. For instance, a cookie r with confidentiality label $C(\lambda(r)) = \mathtt{http}(d) \wedge \mathtt{https}(d)$ is attached both to HTTP and HTTPS requests to domain d. It would be unsound to use a stronger label for typing, e.g., $\mathtt{https}(d)$, since we would miss attacks due to the cookie leakage over HTTP. In the same spirit, we check that URLs do not contain parameters requiring stronger type guarantees than those offered by the type assigned to the URL (1b).

Additionally, well-formedness rules out two inherently insecure type assignments for cookies. First, if a low integrity URL can read a cookie, then the cookie must have low confidentiality since the attacker can inject a script leaking the cookies, as in a typical XSS (2b). Second, cookies that can be set over a low integrity network connection cannot be high confidentiality credentials since the attacker can set them to a value she knows (2c).

C. Intuition Behind the Typing Rules

The type system resembles one for standard information flow control (IFC) where we consider explicit and implicit flows for integrity, but only explicit flows for confidentiality: since our property of interest is web session integrity, regarding confidentiality we are only interested in preventing credentials from being leaked (since they are used for access control), while the leakage of other values does not impact our property. The type system restricts the operations on credentials to be equality checks, hence the leak of information through implicit flows is limited to one bit: this is consistent with the way credentials are handled by real web applications. A treatment of implicit flows for confidentiality would require a declassification mechanism to handle the bit leaked by credential checks, thus complicating our formalism without adding any tangible security guarantee.

As anticipated in Section II, the code is type-checked twice under different assumptions: first, we consider the case of an honest user visiting the server; second, we consider a CSRF attempt where the attacker forces the user's browser to send a request to the server. We do not consider the case of the attacker visiting the server from her own browser since we can prove that such a session is always well-typed, which is close in spirit to the opponent typability lemma employed in type systems for cryptographic protocols [26], [4].

To enforce our session integrity property, the type system needs to track the identity of the user owning the session and the intention of the user to perform authenticated actions. In typing, this is captured by two dedicated labels.

The *session label* ℓ_s records the owner of the active session and is used to label references in the session memory. The label typically equals the one of the session identifier, thus it changes when we resume or start a new session. Formally, $\ell_s \in \mathcal{L}^2 \cup \{\times\}$ where $\times$ denotes no active session.

The *program counter label* $\mathrm{pc} \in \mathcal{L}$ tracks the integrity of the control flow. A high pc implies that the control flow is intended by the user. The pc is lowered in conditionals with a low integrity guard, as is standard in IFC type systems. In the CSRF typing branch, the pc will be permanently low: we need to prune this typing branch to type-check high integrity actions. For this purpose, we use token or origin checks: in the former, the user submits a CSRF token that is compared to a (secret) session reference or cookie, while in the latter we check whether the origin of the request is contained in a whitelist. There are cases in which we statically know that the check will fail, allowing us to prune typing branches.

We also briefly comment on another important attack, namely cross-site scripting (XSS): we can model XSS vulnerabilities by including a script from an attacker-controlled domain, which causes a failure in typing. However, XSS prevention is orthogonal to the goal of our work and must be solved with alternative techniques, e.g., proper input filtering or CSP [34].

D. Explanation of the Typing Rules

1) Server Expressions: typing of server expressions is ruled by the judgement $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau$, meaning that the expression se has type τ in the typing environment Γ within the session ℓ_s. Names have type $\mathtt{cred}(\ell)$ where ℓ is the label provided as an annotation (T-EName, T-EFresh). Values different from names are constants of type $\perp$, i.e., they have low confidentiality and high integrity (T-EVal). Rule (T-EUndef) gives any type to the undefined value $\perp$. This is needed since the initial memory and empty parameters contain this value and have to be well-typed. Types for variables and references in the global memory are read from the corresponding environments (T-EVar,T-EGlobRef). For session references we combine the information stored in the environment with the session label ℓ_s, which essentially acts as an upper bound on the types of references (T-ESesRef). In a honest session, ℓ_s has high confidentiality, thus the session memory can be used to store secrets. In the attacker session, instead, the types of all session references are lowered to at most ℓ_a. Typing fails if no session is active, i.e., $\ell_s = \times$. The computed type for a reference is a credential type if and only if it is so in the environment. Binary operations are given the join of the types of the two operands (T-EBinOp). However, on credentials we allow only equality checks to limit leaks through implicit flows. Finally, (T-ESub) lets us use subtyping on expressions.

2) Server References: typing of server references is ruled by the judgment $\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau)$ meaning that the reference r has type $\mathtt{ref}(\tau)$ in the typing environment Γ within the session ℓ_s. This judgement is used to derive the type of a reference we write into, in contrast to the typing of expressions which covers the typing of references from which we read. While (T-RGlobRef) just looks up the type of the global reference in the typing environment, in (T-RSesRef)

TABLE III: Type system.

Server expressions

(T-ENAME)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} n^{\ell} : \mathtt{cred}(\ell)}$$

(T-EFRESH)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \mathit{fresh}()^{\ell} : \mathtt{cred}(\ell)}$$

(T-EVAL)
$$\frac{v \notin \mathcal{N}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} v : \bot}$$

(T-EUNDEF)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \bot : \tau}$$

(T-EVAR)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \Gamma_{\mathcal{X}}(x)}$$

(T-EGLOBREF)
$$\frac{\Gamma_{\mathcal{R}^{@}}(r) = \mathtt{ref}(\tau)}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} @r : \tau}$$

(T-ESESREF)
$$\frac{\begin{array}{c}\Gamma_{\mathcal{R}^{\$}}(r) = \mathtt{ref}(\tau') \qquad \ell = (C(\tau') \sqcap_C C(\ell_s), I(\tau') \sqcup_I I(\ell_s)) \\ \ell_s \neq \times \qquad \tau = (\tau' \neq \mathtt{cred}(\cdot))\ ?\ \ell\ :\ \mathtt{cred}(\ell)\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \$r : \tau}$$

(T-EBINOP)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se' : \tau' \\ (\tau, \tau' \neq \mathtt{cred}(\cdot)) \vee \odot \text{ is } =\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se \odot se' : \tau \sqcup \tau'}$$

(T-ESUB)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau' \qquad \tau' \sqsubseteq_{\ell_a} \tau}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau}$$

Server references

(T-RGLOBREF)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} @r : \Gamma_{\mathcal{R}^{@}}(r)}$$

(T-RSESREF)
$$\frac{\begin{array}{c}\ell_s \neq \times \qquad \Gamma_{\mathcal{R}^{\$}}(r) = \mathtt{ref}(\tau') \qquad \ell = (C(\tau') \sqcap_C C(\ell_s), I(\tau') \sqcup_I I(\ell_s)) \\ \tau = (\tau' \neq \mathtt{cred}(\cdot))\ ?\ \ell\ :\ \mathtt{cred}(\ell)\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} \$r : \mathtt{ref}(\tau)}$$

(T-RSUB)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau') \qquad \tau \sqsubseteq_{\ell_a} \tau'}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau)}$$

Server-side commands

(T-SKIP)
$$\frac{}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{skip} : \ell_s, \mathtt{pc}}$$

(T-SEQ)
$$\frac{\begin{array}{c}\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s', \mathtt{pc}' \\ \Gamma, \ell_s', \mathtt{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c' : \ell_s'', \mathtt{pc}''\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c; c' : \ell_s'', \mathtt{pc}''}$$

(T-LOGIN)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_u : \tau \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_{pw} : \mathtt{cred}(\ell) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_{sid} : \mathtt{cred}(\ell') \\ C(\mathtt{cred}(\ell)) \sqsubseteq_C C(\mathtt{cred}(\ell')) \qquad I(\mathtt{cred}(\ell)) \sqcup_I \mathtt{pc} \sqsubseteq_I I(\mathtt{cred}(\ell'))\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{login}\ se_u, se_{pw}, se_{sid} : \ell_s, \mathtt{pc}}$$

(T-START)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \mathtt{cred}(\ell) \\ \ell_s' = (C(\mathtt{cred}(\ell)) \sqsubseteq_C C(\ell_a))\ ?\ \ell_a : \ell\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{start}\ se : \ell_s', \mathtt{pc}}$$

(T-SETGLOBAL)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} @r : \mathtt{ref}(\tau) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \\ \mathtt{pc} \sqsubseteq_I I(\tau)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} @r := se : \ell_s, \mathtt{pc}}$$

(T-SETSESSION)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} \$r : \mathtt{ref}(\tau) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \\ \mathtt{pc} \sqsubseteq_I I(\tau)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \$r := se : \ell_s, \mathtt{pc}}$$

(T-IF)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \mathtt{pc}' = \mathtt{pc} \sqcup_I I(\tau) \qquad \Gamma, \ell_s, \mathtt{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s'', \mathtt{pc}_1 \\ \Gamma, \ell_s, \mathtt{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c' : \ell_s''', \mathtt{pc}_2 \qquad \mathtt{pc}'' = \mathtt{pc}_1 \sqcup_I \mathtt{pc}_2 \qquad \ell_s' = (\ell_s'' = \ell_s''')\ ?\ \ell_s'' : \times\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c' : \ell_s', \mathtt{pc}''}$$

(T-AUTH)
$$\frac{\begin{array}{c}\ell_s \neq \times \qquad \forall k \in [1 \ldots |\vec{se}|].\, \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \\ \Big(I(\ell_a) \sqsubseteq_I \bigsqcup_{I\, 1 \leq k \leq |\vec{se}|} I(\tau_k) \sqcup_I \mathtt{pc} \sqcup_I I(\ell_s)\Big) \Rightarrow I(\ell_a) \sqsubseteq_I I(\ell)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell : \ell_s, \mathtt{pc}}$$

(T-PRUNETCHK)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\mathtt{cred}(\ell)) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \tau \\ C(\tau) \neq C(\mathtt{cred}(\ell)) \qquad C(\mathtt{cred}(\ell)) \not\sqsubseteq_C C(\ell_a) \qquad b = \mathsf{csrf}\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, (u,b,\mathcal{P})} \mathbf{if}\ \mathbf{tokenchk}(x, r)\ \mathbf{then}\ c : \ell_s, \mathtt{pc}}$$

(T-TCHK)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\mathtt{cred}(\ell)) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \mathtt{cred}(\ell) \\ \Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s', \mathtt{pc}\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{if}\ \mathbf{tokenchk}(x, r)\ \mathbf{then}\ c : \ell_s', \mathtt{pc}}$$

(T-PRUNEOCHK)
$$\frac{\forall l \in L. I(\ell_a) \not\sqsubseteq_I l \qquad u \in \mathcal{P} \qquad b = \mathsf{csrf}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, (u,b,\mathcal{P})} \mathbf{if}\ \mathbf{originchk}(L)\ \mathbf{then}\ c : \ell_s, \mathtt{pc}}$$

(T-OCHK)
$$\frac{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s', \mathtt{pc}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{if}\ \mathbf{originchk}(L)\ \mathbf{then}\ c : \ell_s', \mathtt{pc}}$$

(T-REPLY)
$$\frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \\ \mathtt{pc}' = \mathtt{pc} \sqcup_I l_r \qquad \Gamma'_{\mathcal{X}} = x_1 : \tau_1, \ldots, x_{|\vec{se}|} : \tau_{|\vec{se}|} \qquad \Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}) \qquad \forall k \in [1 \ldots |\vec{se}|].\, \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \wedge C(\tau_k) \sqsubseteq_C C(\ell_u) \\ \forall r \in \mathit{dom}(ck).\, \Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau_r) \wedge \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} ck(r) : \tau_r \wedge \mathtt{pc}' \sqsubseteq_I I(\tau_r) \qquad \Gamma', b, \mathtt{pc}' \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} s \qquad b = \mathsf{csrf} \Rightarrow \forall x \in vars(s).\, C(\Gamma'_{\mathcal{X}}(x)) \sqsubseteq_C C(\ell_a) \\ b = \mathsf{hon} \Rightarrow \mathtt{pc} \sqsubseteq_I l_r \wedge \Big(page = \mathsf{error} \vee \forall v \in \mathit{dom}(page).\, \Gamma', v, \mathtt{pc}' \vdash^{\mathsf{f}}_{\ell_a} page(v)\Big) \qquad I(\ell_a) \sqsubseteq_I I(\ell_u) \Rightarrow \forall k \in [1 \ldots |\vec{se}|].\, C(\tau_k) \sqsubseteq_C C(\ell_a)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, (u,b,\mathcal{P})} \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} : \ell_s, \mathtt{pc}}$$

(T-REDIR)
$$\frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad \Gamma'_{\mathcal{X}} = x_1 : \tau_1, \ldots, x_{|\vec{se}|} : \tau_{|\vec{se}|} \qquad \Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}) \\ \forall k \in [1 \ldots |\vec{se}|].\, \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \wedge C(\tau_k) \sqsubseteq_C C(\ell_u) \qquad \forall r \in \mathit{dom}(ck).\, \Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau_r) \wedge \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} ck(r) : \tau_r \wedge \mathtt{pc} \sqsubseteq_I I(\tau_r) \\ I(\ell_a) \sqsubseteq_I I(\ell_u) \Rightarrow \forall k \in [1 \ldots |\vec{se}|].\, C(\tau_k) \sqsubseteq_C C(\ell_a) \qquad b = \mathsf{csrf} \Rightarrow \forall x \in vars(\vec{z}).\, C(\Gamma'_{\mathcal{X}}(x)) \sqsubseteq_C C(\ell_a) \\ \Gamma_{\mathcal{U}}(u') = \ell'_u, \vec{\tau'}, \_ \qquad b = \mathsf{hon} \Rightarrow \Big(\mathtt{pc} \sqsubseteq_I I(\ell_u) \wedge m = |\vec{z}| = |\vec{\tau'}| \wedge \forall k \in [1 \ldots m].\, \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} z_k : \tau'_k \wedge \tau'_k \sqsubseteq_{\ell_a} \tau_k\Big)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a, (u,b,\mathcal{P})} \mathbf{redirect}\ (u', \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se} : \ell_s, \mathtt{pc}}$$

Forms

(T-FORM)
$$\frac{\Gamma_{\mathcal{V}}(v) = \Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad \mathtt{pc} \sqsubseteq_I I(\ell_u) \qquad m = |\vec{z}| = |\vec{\tau}| \qquad \forall k \in [1 \ldots m].\, \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} z_k : \tau'_k \wedge \tau'_k \sqsubseteq_{\ell_a} \tau_k}{\Gamma, v, \mathtt{pc} \vdash^{\mathsf{f}}_{\ell_a} \mathsf{form}(u, \vec{z})}$$

Server threads

(T-PARALLEL)
$$\frac{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a, \mathcal{P}} t \qquad \Gamma^0 \vdash^{\mathsf{t}}_{\ell_a, \mathcal{P}} t'}{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a, \mathcal{P}} t \parallel t'}$$

(T-RECV)
$$\frac{\begin{array}{c}\lambda, \ell_a, \Gamma^0 \vdash \diamond \qquad \Gamma^0_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad m = |\vec{\tau}| = |\vec{x}| \\ \forall k \in [1 \ldots |\vec{r}|].\, C(\Gamma^0_{\mathcal{R}^{@}}(r_k)) \sqsubseteq_C C(\ell_u) \wedge I(\ell_u) \sqsubseteq_I I(\Gamma^0_{\mathcal{R}^{@}}(r_k)) \\ \Gamma_{\mathcal{X}} = x_1 : \tau_1, \ldots, x_m : \tau_m \qquad (\Gamma^0_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^{@}}, \Gamma^0_{\mathcal{R}^{\$}}, \Gamma^0_{\mathcal{V}}), \times, I(\ell_u) \vdash^{\mathsf{c}}_{\ell_a, (u, \mathsf{hon}, \mathcal{P})} c : \_, I(\ell_u) \\ \Gamma'_{\mathcal{X}} = x_1 : \ell_a, \ldots, x_m : \ell_a \qquad (\Gamma^0_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^{@}}, \Gamma^0_{\mathcal{R}^{\$}}, \Gamma^0_{\mathcal{V}}), \times, \top_I \vdash^{\mathsf{c}}_{\ell_a, (u, \mathsf{csrf}, \mathcal{P})} c : \_, \top_I\end{array}}{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a, \mathcal{P}} u[\vec{r}](\vec{x}) \hookrightarrow c}$$

we have analogous conditions to (T-ESESREF) for session references. Subtyping for reference types is contra-variant to subtyping for security types (T-RSUB).

3) Server Commands: the judgement $\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : {\ell_s}', \mathtt{pc}'$ states that the command c (bound to the endpoint at URL u) can be typed against the attacker ℓ_a in the typing branch $b \in \{\mathsf{hon}, \mathsf{csrf}\}$ using typing environment Γ, session label ℓ_s and program counter label $\mathtt{pc}$. $\mathcal{P}$ contains all URLs that rely on an origin check to prevent CSRF attacks. After the execution of c, the session label and the PC label are respectively updated to ${\ell_s}'$ and $\mathtt{pc}'$. We let $C = (u, b, \mathcal{P})$ if the individual components of the tuple are not used in a rule. The branch b tracks whether we are typing the scenario of an honest request ($b = \mathsf{hon}$) or the CSRF case ($b = \mathsf{csrf}$).

Rule (T-SKIP) does nothing, while (T-SEQ) types the second command with the session label and the PC label obtained by typing the first command.

Rule (T-LOGIN) verifies that the password and the session identifier are both credentials and that the latter is at least as confidential as the former, since the identifier can be used for authentication in place of the password. Finally, we check that the integrity of the password and $\mathtt{pc}$ are at least as high as the integrity of the session identifier to prevent an unauthorized party from influencing the identity associated to the session.

Rule (T-START) updates the session label used for typing the following commands. First we check that the session identifier se has a credential type: if it has low confidentiality, we update the session label to ℓ_a (since the attacker can access the session), otherwise we use the label ℓ in the type of se.

Rules (T-SETGLOBAL) and (T-SETSESSION) ensure that no explicit flow violates the confidentiality or integrity policies, where for integrity we also consider the PC label.

Rule (T-IF) lowers the PC based on the integrity label of the guard expression of the conditional and uses it to type-check the two branches. In the continuation we use the join of the PC labels returned in the two branches: using a higher PC would be unsound since reaching the continuation may depend on which branch is executed (e.g., if one of the branches contains a **reply** command). If typing the two branches yields two different session labels, we use the session label $\times$ in the continuation to signal that the session state cannot be statically predicted and thus no session operation should be allowed.

Rule (T-AUTH) ensures that the attacker cannot affect any component leading to an authenticated event (PC label, session label or any expression in $\vec{se}$) unless the event is annotated with a low integrity label. Since authenticated events are bound to sessions, we require $\ell_s \neq \times$.

Rules (T-PRUNETCHK) and (T-TCHK) handle CSRF token checks. In (T-PRUNETCHK) we statically know that the check fails since the reference where the token is stored has a high confidentiality credential type and the parameter providing the token is a low confidentiality value, hence we do not type-check the continuation c. This reasoning is sound since credentials are unguessable fresh names and we disallow subtyping for high confidentiality credentials, i.e., public values cannot be treated as secret credentials. This rule is used only in the CSRF typing branch. Rule (T-TCHK) covers the case where the check may succeed and we simply type-check the continuation c. We do not change the PC label since a failure in the check produces an error page which causes the user to stop browsing.

Similarly, rules (T-PRUNEOCHK) and (T-OCHK) cover origin checks. We can prune the CSRF typing branch if the URL we are typing is protected ($u \in \mathcal{P}$) and all whitelisted origins have high integrity, since the origin of a CSRF attack to a protected URL has always low integrity.

Rule (T-REPLY) combines the PC label with the expected integrity label of the response l_r for the current URL to compute $\mathtt{pc}'$ which is used to type the response. In the honest typing branch, we require $\mathtt{pc}' = l_r$, which establishes an invariant used when typing an **include** command in a browser script, where we require that the running script and the included script can be typed with the same $\mathtt{pc}$ (cf. rule (T-BINCLUDE) in [15]). Using the typing environment Γ' which contains types for the variables embedded in the response, we check the following properties:

- secrets are not disclosed over a network connection which cannot guarantee their confidentiality;
- the types of the values assigned to cookies are consistent with those in the typing environment (where the PC label is taken into account for the integrity component);
- the script in the response is well-typed (rules in [15]);
- secrets are not disclosed to a script in the CSRF typing branch since it might be included by an attacker's script;
- in the honest typing branch, we check that the returned page is either the error page or all its forms are well-typed according to rule (T-FORM). We do not perform this check in the CSRF branch since a CSRF attack is either triggered by a script inclusion or through a redirect. In the first case the attacker cannot access the DOM, which in a real browser is enforced by the Same Origin Policy. In the second case, well-formed user behavior (cf. Definition 4) ensures that the user will not interact with the DOM in this scenario;
- no high confidentiality data is included in replies over a low integrity network connection, since the attacker could inject scripts to leak secrets embedded in the response.

Rule (T-REDIR) performs mostly the same checks as (T-REPLY). Instead of typing script and DOM, we perform checks on the URL similar to the typing of forms, as discussed below.

4) Forms: the judgement $\Gamma, v, \mathtt{pc} \vdash^{\mathsf{f}}_{\ell_a} f$ says that a form f identified by the name v is well-typed in the environment Γ under the label $\mathtt{pc}$. Our rule for typing forms (T-FORM) first checks that the type of the form name matches the type of the target URL. This is needed since for well-formed user behavior (cf. Definition 4) we assume that the user relies on the name of a form to ensure that her inputs are compliant with the expected types. With $\mathtt{pc} \sqsubseteq_I I(\ell_u)$ we check that the thread running with program counter label $\mathtt{pc}$ is allowed to trigger requests to u. In this way we can carry over the $\mathtt{pc}$ from one thread where the form has been created to the one receiving the request since we type-check the honest branch with $\mathtt{pc} = I(\ell_u)$. Finally, we check that the types of form values comply with the expected type for the corresponding URL parameters, taking the PC into account for implicit integrity flows.

5) Server Threads: the judgement $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$ says that the thread t is well-typed in the environment Γ^0 against the attacker ℓ_a and $\mathcal{P}$ is the set of URLs protected against CSRF attacks via origin checking.

Rule (T-PARALLEL) states that the parallel composition of two threads is well-typed if both are well-typed. Rules for typing running threads (i.e., $t = \lceil c \rfloor^R_E$) are in the technical report [15], since they are needed only for proofs.

Rule (T-RECV) checks that the environment is well-formed and that the network connection type ℓ_u is strong enough to guarantee the types of the cookies, akin to what is done for parameters in Definition 3. Then we type-check the command twice with $\ell_s = \times$, since no session is initially active. In the first branch we let $b = \mathsf{hon}$: parameters are typed according to the type of u in $\Gamma^0_{\mathcal{U}}$ which is reflected in the environment $\Gamma_{\mathcal{X}}$. As the honest user initiated the request, we let $\mathtt{pc} = I(\ell_u)$, i.e., we use the integrity label of the network connection as $\mathtt{pc}$. This allows us to import information about the program counter from another (well-typed) server thread or browser script that injected the form into the DOM or directly triggered the request. In the second branch we let $b = \mathsf{csrf}$: parameters are chosen by the attacker, hence they have type ℓ_a in $\Gamma'_{\mathcal{X}}$. As the attacker initiated the request, we let $\mathtt{pc} = \top_I$.

E. Formal Results

We introduce the notion of *navigation flow*, which identifies a sequence of navigations among different pages occurring in a certain tab and triggered by the user's interaction with the elements of the DOM of rendered pages. Essentially, a navigation flow is a list of user actions consisting of a load on a certain tab followed by all actions of type submit in that tab (modeling clicks on links and submissions of forms) up to the next load (if any). A formal definition is presented in [15].

Next we introduce the notion of *well-formedness* to constrain the interactions of an honest user with a web system.

Definition 4. *The list of user actions $\vec{a}$ is* well-formed *for the honest user* usr *in a web system W with respect to a typing environment Γ^0 and an attacker ℓ_a iff*

1) for all actions a' in $\vec{a}$ we have:
- *if $a' = \mathsf{load}(tab, u, p)$, $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ then for all $k \in dom(p)$ we have $p(k) = v^{\tau'} \Rightarrow \tau' \sqsubseteq_{\ell_a} \tau_k$;*
- *if $a' = \mathsf{submit}(tab, u, v', p)$, $\Gamma_{\mathcal{V}}(v') = \ell_u, \vec{\tau}, l_r$ then for all $k \in dom(p)$ we have $p(k) = v^{\tau'} \Rightarrow \tau' \sqsubseteq_{\ell_a} \tau_k$.*

2) $(\ell_a, \mathcal{K}_0) \triangleright B_{\mathsf{usr}}(\{\}, \{\}, \vec{a}) \parallel W \xrightarrow{\gamma}{}^{*} (\ell_a, \mathcal{K}') \triangleright B_{\mathsf{usr}}(M, P, \langle\rangle) \parallel W'$ for some $\mathcal{K}', W', M, P$ where γ is an unattacked trace;

3) for every navigation flow $\vec{a}'$ in $\vec{a}$, we have that $I(\ell_a) \sqsubseteq_I I(\lambda(a'_j))$ implies $I(\ell_a) \sqsubseteq_I I(\lambda(a'_k))$ for all $j < k \leq |\vec{a}'|$.

Condition 1 prevents the user from deliberately leaking secrets by enforcing that the expected parameter types are respected. While the URL in a load event is the target URL and we can directly check its type, in a submit action it refers to the page containing the form: intuitively, this models a user who knows which page she is actively visiting with a load and which page she is currently on when performing a submit. However, we do not expect the user to inspect the target URL of a form. Instead, we expect the user to identify a form by its displayed name (the parameter v' in submit) and input only data matching the type associated to that form name. For instance, in a form named "public comment", we require that the user enters only public data. Typing hence has to enforces that all forms the user interacts with are named correctly. Otherwise, an attacker could abuse a mismatch of form name and target URL in order to steal confidential data.

Condition 2 lets us consider only honest runs in which the browser terminates regularly. Concretely, this rules out interactions that deliberately trigger an error at the server-side, e.g., the user loads a page expecting a CSRF token without providing this token, or executions that do not terminate due to infinite loops, e.g., where a script recursively includes itself.

Condition 3 requires that the user does not navigate a trusted website reached by interacting with an untrusted page. Essentially, this rules out phishing attempts where the attacker influences the content shown to the user in the trusted website.

Our security theorem predicates over *fresh clusters*, i.e., systems composed of multiple servers where no command is running or has been run in the past.

Definition 5. *A server S is* fresh *if $S = (\{\}, \{\}, t)$ where t is the parallel composition of threads of the type $u[\vec{r}](\vec{x}) \hookrightarrow c$. A system W is a* fresh cluster *if it is the parallel composition of fresh servers.*

We now present the main technical result, namely that well-typed clusters preserve the session integrity property from Definition 2 for all well-formed interactions of the honest user with the system, provided that her passwords are confidential.

Theorem 1. *Let W be a fresh cluster, $(\ell_a, \mathcal{K})$ an attacker, Γ^0 a typing environment, $\mathcal{P}$ a set of protected URLs against CSRF via origin checking and let $\vec{a}$ be a list of well-formed user actions for* usr *in W with respect to Γ^0 and ℓ_a. Assume that for all u with $\rho(\mathsf{usr}, u) = n^{\ell}$ we have $C(\ell) \not\sqsubseteq_C C(\ell_a)$ and for all $n^{\ell} \in \mathcal{K}$ we have $C(\ell) \sqsubseteq_C C(\ell_a)$. Then W preserves* session integrity *against ℓ_a with knowledge $\mathcal{K}$ for the honest user* usr *performing the list of actions $\vec{a}$ if $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$ for all servers $S = (\{\}, \{\}, t)$ in W.*

The proof builds upon a simulation relation connecting a run of the system with the attacker with a corresponding run of the system without the attacker in which the honest user behaves in the same way and high integrity authenticated events are equal in the two runs. The full security proof can be found in the technical report [15].

V. CASE STUDY

Now we resume the analysis of HotCRP, started in Section II where we described the login CSRF and proposed a fix, and describe the remaining session integrity problems we discovered by typing its model in our core calculus. The encodings of Moodle and phpMyAdmin, including the description of the new vulnerability, are provided in the technical report [15].

A. Methodology

We type-check the HotCRP model of Section II against different attackers, including the web-, related-domain-, and network attacker. Two scenarios motivate the importance of the related-domain attacker in our case study. First, many conferences using HotCRP deploy the system on a subdomain of the university organizing the event, e.g., CSF 2020: any user who can host contents on a subdomain of the university can act as the attacker. Second, anybody can host a conference on a subdomain of hotcrp.com or access the administrative panel of test.hotcrp.com: by exploiting a stored XSS vulnerability (now fixed) in the admin panel, it was possible to show on the homepage of the conference a message containing JavaScript code that tampers with cookies to implement the attacks below.

Failures in type-checking highlight code portions that we analyze manually, as they likely suffer from session integrity flaws. Once a problem is identified, we implement a patch in our HotCRP model and try to type-check it again; this iterative process stops when we manage to establish a security proof by typing, as shown in Section V-C.

B. Cookie Integrity Attacks

Our fix against login CSRF does not ensure the integrity of session cookies against network and related-domain attackers: the former can compromise cookie integrity by forging HTTP traffic, while the latter can set cookies for the target website by using the `Domain` attribute. Attackers can thus perform *cookie forcing* to set the their session cookies in the victim's browser, achieving the same outcome of a login CSRF.

Even worse, the lack of cookie integrity combined with a logical vulnerability on HotCRP code enables a *session fixation* attack, where the attacker manages to force a known cookie into the browser of the victim before she authenticates which is used by HotCRP to identify the victim's session after login. With the known cookie, the attacker can then access the victim's session to steal submitted papers, send fake reviews, or deanonymize reviewers. HotCRP tries to prevent session fixation by checking during login whether the provided session cookie (if any) identifies a session where no variable is set: in such a case, the value of the cookie is changed to an unpredictable random string. However, some session variables are not properly unset during logout, thus the above check can be voided by an attacker with an account on the target website that obtains a valid cookie by authenticating and logging out.[3] At this point, the attacker can inject this cookie into the victim's browser to perform the attack.

Both attacks are captured in typing as follows: although we have a certain liberty in the choice of our initial environment, no possible type for sid leads to a successful type derivation since sid must have a credential type. As the attacker can set the cookie, it must have low integrity by well-formedness of the typing environment (Definition 3). Since the attacker can write (low confidentiality) values of her knowledge into sid, it may not be a credential of high confidentiality, again by Definition 3. Hence we must assume that sid is a credential of low confidentiality and integrity. However, since the user's password has high confidentiality, typing fails in the *login* endpoint (on line 9) when applying rule (T-LOGIN).

A possible solution against these threats relies on the adoption of *cookie prefixes* (cf. Section III-C) which provide high integrity guarantees against network and related-domain attackers. This protection cannot be applied by default in HotCRP due to backward compatibility reasons, i.e., hotcrp.com relies on cookies shared across multiple domains to link different conferences under the same account. However, the developer has fixed the bug causing the session fixation vulnerability and we have discussed with him the option to offer cookie prefixes as an opt-in security mechanism during the setup of HotCRP.

C. Typing Example

Now we show how to type-check the (fixed) *login* endpoint on domain d_C against an attacker controlling a related-domain $d_E \sim d_C$, assuming that the session cookie is secured with the `__Host-` prefix. We let the attacker label $\ell_a = (\mathsf{http}(d_E) \vee \mathsf{https}(d_E), \mathsf{http}(d_E) \vee \mathsf{https}(d_E))$, and let $\ell_C = (\mathsf{https}(d_C), \mathsf{https}(d_C))$, $\ell_{LH} = (\bot_C, \mathsf{https}(d_C))$, $\ell_{HL} = (\mathsf{https}(d_C), \top_I)$. We then consider a minimal environment Γ sufficient to type the *login* endpoint, where:

$$\begin{aligned}
\Gamma_{\mathcal{U}} &= \{\mathit{login} \mapsto (\ell_C, (\ell_{LH}, \mathtt{cred}(\ell_C), \mathtt{cred}(\ell_{HL})), \mathsf{https}(d_C)),\\
&\qquad \mathit{manage} \mapsto (\ell_C, (\ell_C, \ell_{LH}, \mathtt{cred}(\ell_{HL})), \mathsf{https}(d_C))\}\\
\Gamma_{\mathcal{R}^@} &= \{r \mapsto \mathtt{cred}(\ell_C), r' \mapsto \mathtt{cred}(\ell_{HL}),\\
&\qquad \mathit{sid} \mapsto \mathtt{cred}(\ell_C), \mathit{pre} \mapsto \mathtt{cred}(\ell_{HL})\}\\
\Gamma_{\mathcal{R}^\$} &= \{\mathit{user} \mapsto \ell_{LH}, \mathit{ltoken} \mapsto \mathtt{cred}(\ell_{HL})\}\\
\Gamma_{\mathcal{V}} &= \{\mathtt{auth} \mapsto \Gamma_{\mathcal{U}}(\mathit{login}), \mathtt{link} \mapsto \Gamma_{\mathcal{U}}(\mathit{manage})\}
\end{aligned}$$

We typecheck the code under two different assumptions in (T-RECV). Our goal is to prune the CSRF typing branch before the security critical part and type it only in the honest setting.

We start with the honest typing branch. When typing the conditional (line 2) in rule (T-IF), we do not lower pc since the integrity label of the guard and pc is $\mathsf{https}(d_C)$. In the **then** branch (line 3), we have the assignment $@r' := \mathit{fresh}()^{\ell_{HL}}$, which types successfully according to (T-SETGLOBAL). The **start** statement with the freshly sampled value yields a session label $\ell_s = (\mathsf{https}(d_C), \top_I)$. The assignment $\$\mathit{ltoken} := \mathit{fresh}()^{\ell_{HL}}$ also succeeds according to (T-SETSESSION). The session label does not affect the type of the reference $\$\mathit{ltoken}$ in this case. For the **reply** (lines 4–5) we successfully check that the URL is well-formed and may be produced with the current pc (T-FORM), that the empty script is well-typed, and that $y = @r'$ may be assigned to the cookie pre (T-REPLY). In the **else** branch of the conditional, we start a session over the cookie $@\mathit{pre}$

[3] To simplify the presentation (and due to space constraints), this complex behavior is not encoded in the example in Section II. However, the possibility to perform cookie forcing, which is modeled in our example, is a prerequisite for session fixation and is detected by the type system.

(line 7), leading to a session label $\ell_s = (\mathsf{https}(d_C), \top_I)$ (T-START). The conditions in (T-TCHK) are fulfilled for the **tokenchk** command (line 8) and we continue typing without any additional effect. Since we still have $\texttt{pc} = \mathsf{https}(d_C)$, the assignment $@r := \mathit{fresh}()^{\ell_C}$ type-checks (line 9). As the password is of the same type as the reference $@r$ containing the session secret, the **login** also type-checks successfully (T-LOGIN). The **start** statement over a credential of type $\texttt{cred}(\ell_C)$ gives us the session label $\ell_s = \ell_C$. For the **reply** (lines 10–11), we check that we may include the form with the current $\texttt{pc}$ and that it is well formed (trivial since it contains only $\perp$), that the empty script is well-typed and that we may assign the value of $@r$ to the cookie *sid* (T-REPLY).

The **then** branch of the CSRF case types similarly to the honest case, since all references used in it and the cookie *pre* have integrity label $\top_I$. Additionally, in the CSRF branch, we do not type the DOM (T-REPLY). In the **else** branch we start a session (line 7) with label $\ell_s = (\mathsf{https}(d_C), \top_I)$ (T-START). When performing the **tokenchk** (line 8), we can apply rule (T-PRUNETCHK), since $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \$ltoken : \texttt{cred}(\ell_{HL})$ and $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} token : \ell_a$ cannot be given the same confidentiality label. Hence, we do not have to type-check the continuation.

VI. RELATED WORK

Formal foundations for web security have been proposed in a seminal paper [1], using a model of the web infrastructure expressed in the Alloy model-checker to find violations of expected web security goals. Since then, many other papers explored formal methods in web security: a recent survey [11] covers different research lines. We discuss here the papers which are closest to our work.

In the context of web sessions, [12] employed *reactive non-interference* [10] to formalize and prove strong confidentiality properties for session cookies protected with the `HttpOnly` and `Secure` attributes, a necessary condition for any reasonable notion of session integrity. A variant of reactive non-interference was also proposed in [29] to formalize an integrity property of web sessions which rules out CSRF attacks and malicious script inclusions. The paper also introduced a browser-side enforcement mechanism based on *secure multi-execution* [21]. A more general definition of web session integrity, which we adapted in the present paper, was introduced in [13] to capture additional attacks, like password theft and session fixation. The paper also studied a provably sound browser-based enforcement mechanism based on runtime monitoring. Finally, [14] proposed the adoption of *micro-policies* [20] in web browsers to prevent a number of attacks against web sessions and presented Michrome, a Google Chrome extension implementing the approach. None of these papers, however, considered the problem of enforcing a formal notion of session integrity by analyzing web application code, since they only focused on browser-side defenses.

Formal methods found successful applications to web session security through the analysis of *web protocols*, which are the building blocks of web sessions when single sign-on services are available. Bounded model-checking was employed in [3] and [2] to analyze the security of existing single sign-on protocols, exposing real-world attacks against web authentication. WebSpi is a ProVerif library designed to model browser-server interactions, which was used to analyze existing implementations of single sign-on based on OAuth 2.0 [7] and web-based cloud providers [6].

Web protocols for single sign-on have also been manually analyzed in the expressive Web Infrastructure Model (WIM): for instance, [23] focused on OAuth 2.0, [24] considered OpenID Connect, and [22] analyzed the OpenID Financial-grade API. While the WIM is certainly more expressive than our core model, proofs are at present manual and require a strong human expertise. In terms of security properties, [22] considers a session integrity property expressed as a trace property that is specific to the OpenID protocol flow and the resources accessed thereby, while our definition of session integrity is generic and formulated as a hyperproperty.

Server-side programming languages with formal security guarantees have been proposed in several research papers. Examples include SELinks [19], UrFlow [17], SeLINQ [32] and JSLINQ [5]. All these languages have the ability to enforce information flow control in multi-tier web applications, potentially including a browser, a server and a database. Information flow control is an effective mechanism to enforce session integrity, yet these papers do not discuss how to achieve web session security; rather, they propose new languages and abstractions for developing web applications. To the best of our knowledge, there is no published work on the formal security analysis of server-side programming languages, though the development of accurate semantics for such languages [25] is undoubtedly a valuable starting point for this kind of research.

VII. CONCLUSION

We introduced a type system for sound verification of session integrity for web applications encoded in a core model of the web, and used it to assess the security of the session management logic of HotCRP, Moodle, and phpMyAdmin. During this process we unveiled novel critical vulnerabilities that we responsibly disclosed to the applications' developers, validating by typing the security of the fixed versions.

We are currently developing a type-checker to fully automate the analysis, which we intend to make available as open source. Providing type annotations is typically straightforward, as they depend on the web application specification and are easily derivable from it (e.g., cookie labels are derived from their attributes) and typing derivations are mostly deterministic, with a few exceptions (e.g., subtyping) that however follow recurrent patterns (e.g., subtyping is used in assignments to upgrade the value type to the reference type).

Furthermore, while in this work we focused on a concise web model to better illustrate the foundational aspects of our analysis technique, it would be interesting to extend the type system to cover richer web models, e.g., the WIM model [22], as well as additional web security properties. We also plan to automate the verification process for PHP code, e.g., by developing an automated translation from real world code into our calculus. Finally, we would like to formalize our theory in a proof assistant.

Acknowledgments

This work has been partially supported by the the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Austrian Science Fund (FWF) through the project PROFET (grant agreement P31621); by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 SBA.

References

[1] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, 2010, pp. 290–304.

[2] 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*, vol. 33, pp. 41–58, 2013.

[3] 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 *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, 2008, pp. 1–10.

[4] M. Backes, C. Hriţcu, and M. Maffei, "Union, Intersection and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations," *Journal of Computer Security*, vol. 22, pp. 301–353, 2014.

[5] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "JSLINQ: Building Secure Applications across Tiers," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, CODASPY 2016*, 2016, pp. 307–318.

[6] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage," in *Proceedings of the 2nd International Conference on Principles of Security and Trust, POST 2013*, 2013, pp. 126–146.

[7] ——, "Discovering Concrete Attacks on Website Authorization by Formal Analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

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

[9] A. Barth, C. Jackson, and J. C. Mitchell, "Robust Defenses for Cross-Site Request Forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*, 2008, pp. 75–88.

[10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive Noninterference," in *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009*, 2009, pp. 79–90.

[11] M. Bugliesi, S. Calzavara, and R. Focardi, "Formal methods for web security," *Journal of Logic and Algebraic Programming*, vol. 87, pp. 110–126, 2017.

[12] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "CookiExt: Patching the Browser Against Session Hijacking Attacks," *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[13] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta, "Provably Sound Browser-Based Enforcement of Web Session Integrity," in *Proceedings of the 27th IEEE Computer Security Foundations Symposium, CSF 2014*, 2014, pp. 366–380.

[14] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei, "Micro-Policies for Web Session Security," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium, CSF 2016*, 2016, pp. 179–193.

[15] S. Calzavara, R. Focardi, N. Grimm, M. Maffei, and M. Tempesta, "Language-Based Web Session Integrity," *arXiv e-prints*, p. arXiv:2001.10405, Jan 2020.

[16] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the Web: A Journey into Web Session Security," *ACM Computing Surveys*, vol. 50, no. 1, pp. 13:1–13:34, 2017.

[17] A. Chlipala, "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, 2010, pp. 105–118.

[18] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, September 2010.

[19] B. J. Corcoran, N. Swamy, and M. W. Hicks, "Cross-Tier, Label-Based Security Enforcement for Web Applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009*, 2009, pp. 269–282.

[20] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P 2015*, 2015, pp. 813–830.

[21] D. Devriese and F. Piessens, "Noninterference through Secure Multi-execution," in *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*, 2010, pp. 109–124.

[22] D. Fett, P. Hosseyni, and R. Küsters, "An Extensive Formal Security Analysis of the OpenID Financial-Grade API," in *Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P 2019*, 2019, pp. 453–471.

[23] D. Fett, R. Küsters, and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*, 2016, pp. 1204–1215.

[24] ——, "The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines," in *Proceedings of the 30th IEEE Computer Security Foundations Symposium, CSF 2017*, 2017, pp. 189–202.

[25] D. Filaretti and S. Maffeis, "An Executable Formal Semantics of PHP," in *Proceedings of the 28th European Conference in Object-Oriented Programming, ECOOP 2014*, 2014, pp. 567–592.

[26] R. Focardi and M. Maffei, *Types for Security Protocols*. IOS Press, 2011, pp. 143–181.

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

[28] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing Cross Site Request Forgery Attacks," in *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, SecureComm 2006*, 2006, pp. 1–10.

[29] W. Khan, S. Calzavara, M. Bugliesi, W. D. Groef, and F. Piessens, "Client Side Web Session Integrity as a Non-interference Property," in *Proceedings of the 10th International Conference on Information Systems Security, ICISS 2014*, 2014, pp. 89–108.

[30] MITRE, "CVE-2019-12616," June 2019. [Online]. Available: https://www.cvedetails.com/cve/CVE-2019-12616/

[31] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, "SessionShield: Lightweight Protection against Session Hijacking," in *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011*, 2011, pp. 87–100.

[32] D. Schoepe, D. Hedin, and A. Sabelfeld, "SeLINQ: Tracking Information Across Application-Database Boundaries," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014*, 2014, pp. 25–38.

[33] 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*, 2011, pp. 615–626.

[34] W3C, "Content Security Policy Level 2," December 2016. [Online]. Available: https://www.w3.org/TR/CSP2/

[35] M. West, "Cookie Prefixes." [Online]. Available: https://tools.ietf.org/html/draft-west-cookie-prefixes-05

[36] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies Lack Integrity: Real-World Implications," in *Proceedings of the 24th USENIX Security Symposium, USENIX Security 2015*, 2015, pp. 707–721.