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 them 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 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. Other common attacks include cross-site request forgery (CSRF) 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. We refer to a recent survey for an overview of attacks against web sessions and countermeasures.

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 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.

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. 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, as a semantic hyperproperty 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. 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 command expects a username $$uid$$ and a password $$pwd$$ used for authentication:

1. login $$[(uid, pwd)]$$
2. if $$uid = \bot$$ and $$pwd = \bot$$ then
3. reply $$[\{\text{auth} \mapsto \text{form}([\text{login}, \{\bot, \bot\}]), \text{skip}, \{\}\}]$$
4. else
5. $$@r := \text{fresh}();$$ login $$uid, pwd, @r;$$
6. start $$@r;$$ $$_user := uid;$$
7. reply $$[\{\text{link} \mapsto \text{form}([\text{manage}, \{\bot, \bot, \bot\}]),$$
8. $$\text{skip}, \{\text{sid} \mapsto x\}\}]$$
9. 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 $$@r$$ (lines 5–6).

For technical convenience, in the login command we also specify the fresh session identifier as a third parameter to bind the identity of its owner to the session. 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 6). 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 7–9).

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

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 5–10). 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 11) and performs the requested action upon success (line 12). 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 $$\ell_C$$, intuitively meaning that authorization can be trusted unless the attacker can read or write at $$\ell_C$$. 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 $$\ell_C = (\text{https}(d_C), \text{https}(d_C))$$, 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 \( \ell_E = (https(d_E), https(d_E)) \). Since \( \ell_E \not\supseteq \ell_C \), 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 \( d_E \):

\[
\begin{align*}
\tilde{a} &= \text{load}(1, \text{login}, \{\}) , \\
&\quad \text{submit}(1, \text{login}, \text{auth}, \{1 \mapsto \text{usr}, 2 \mapsto \text{pwd}\}) , \\
&\quad \text{load}(2, \text{evil}, \{\}) , \\
&\quad \text{submit}(1, \text{manage}, \text{add}, \{1 \mapsto \text{paper}\})
\end{align*}
\]

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 \( \#\text{paper,submit}_{\ell_C,\text{usr}} \), 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 \( \text{auth, paper, action at } \ell_C \) 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 11). When typing, if we assume (in the typing environment) that \( \$\text{token} \) 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.

In the honest scenario, the PC label has high integrity assuming that all the preceding conditionals have high integrity guard expressions (lines 3 and 5). The session label is set in the command start \( @\text{sid} \) (line 2) and depends on the type of the session identifier \( @\text{sid} \). To succeed in typing, \( @\text{sid} \) 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

\[1\] This reasoning is sound only when credentials (e.g., session identifiers and CSRF tokens) are unguessable fresh names. To take into account this aspect, in the type system we have special types for credentials (cf. Section IV.2) and we forbid subtyping for high confidentiality credentials.
entire code must be typed with a low integrity PC label. This prevents typing the reply statement where cookie $sid$ is set (lines 7–9), 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 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. $\text{login}_{\text{pre}}(\text{uid}, \text{pwd}, \text{token}) \Rightarrow$
2. if $\text{uid} = \bot$ and $\text{pwd} = \bot$ then
3. $\alpha r' := \text{fresh}(); \text{start} \alpha r'$; $\$\text{token} := \text{fresh}();$
4. reply $\{\text{auth} \mapsto \text{form}((\text{login}, (\bot, \bot, x)))\}$,
5. skip, $\{\text{pre} \mapsto y\}$
6. with $x = \$\text{token}, y = \alpha r'$
7. else
8. start $\alpha pre$;
9. if $\text{tokenchk}(\text{token}, \$\text{token})$ then
10. $\alpha r := \text{fresh}(); \text{login} \text{uid}, \text{pwd}, \alpha r$;
11. start $\alpha r$; $\$\text{user} := \text{uid}$;
12. reply $\{\text{link} \mapsto \text{form}((\text{manage}, (\bot, \bot, \bot)))\}$,
13. skip, $\{\text{sid} \mapsto x\}$
14. with $x = \alpha r$

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 $\$\text{token}$ 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 $\text{token}$ is compared against the high confidentiality session reference $\$\text{token}$ of the session identified by $\alpha pre$ (line 9). 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 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 [40] 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 $\vec{r}' := \vec{r}$ 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 $\text{dom}(M)$ be the domain of $M$ and $\emptyset$ be the empty map. Given two maps $M_1$ and $M_2$, we define $M_1 \cup M_2$ as the map $M$ such that $M(k) = v$ iff either $M_2(k) = v$ or $k \notin \text{dom}(M_2)$ and $M_1(k) = v$. We write $M_1 \bowtie M_2$ to denote $M_1 \cup M_2$ 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 $\ell$, 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. $\vec{I}$ is the set of identities representing users: we distinguish a special identity $\text{usr}$ representing the honest user and we assume that the other identities are under the attacker’s control.

A URL $u$ is a triple $(\pi, d, v)$ where $\pi \in \{\text{http}, \text{https}\}$ 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 $\perp$ for a blank value.

We let $v$ range over values, i.e., names, primitive values (booleans, integers, etc.), URLs, identities and the blank value $\perp$. 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 $\text{form}(u, z)$, where $u$ is the target URL and $z$ is the list of parameters provided via the query string of a link or in the HTTP body of the request for forms.

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 $M(r) = \perp$ if $r \notin \text{dom}(M)$, 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 $\ell$), 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 $\text{login } se_{ua}, se_{pw}, se_{id}$ models a login operation with username $se_{ua}$ and password $se_{pw}$. The identity of the user is bound to the session identifier obtained by evaluating $se_{id}$. Command $\text{start } se$ starts a new session or restores a previous one identified by the value of the expression $se$. Command $\text{auth } \ell se$ at $\ell$ produces an authenticated event that includes data identified by the list of expressions $\ell se$. The command is annotated with a label $\ell$ denoting the expected security level of the event which has a central role in the security definition presented in Section III-E.

### Table I: Syntax (browsers $B$ and scripts $s$ are defined in appendix A).

<table>
<thead>
<tr>
<th>Basics</th>
<th>Command $c ::= \text{skip} \mid \text{halt} \mid c' \mid @r ::= se \mid $r ::= se$</th>
</tr>
</thead>
<tbody>
<tr>
<td>References $r \in \mathbb{R}$</td>
<td>if $se$ then $c$ else $c'$ \text{ login } se_{ua}, se_{pw}, se_{id}$</td>
</tr>
<tr>
<td>Domains $d \in \mathbb{D}$</td>
<td>$\text{start } se$ at $\ell$</td>
</tr>
<tr>
<td>Simple labels $l \in \mathbb{L}$</td>
<td>if tokenchk($e, e'$) then $c$</td>
</tr>
<tr>
<td>Numbers $k, m \in \mathbb{N}$</td>
<td>if originchk($L$) then $c$</td>
</tr>
<tr>
<td>Forms $f ::= { } \mid f \bowtie { v \mapsto \text{form}(u, z)}$</td>
<td>reply($page, s, ck$) with $\vec{x} = \vec{s}$</td>
</tr>
<tr>
<td>Cookies $ck ::= { } \mid ck \bowtie { r \mapsto z}$</td>
<td>redirect($u, \vec{z}, ck$) with $\vec{x} = \vec{s}$</td>
</tr>
</tbody>
</table>

**User behavior**

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

**Web Systems**

| Server's Knowledge $K \subseteq \mathbb{N}$ | $W ::= B \mid S \mid \mathbb{W}$ |
| Web Systems $W$ | $\text{Attacked Systems } A ::= (\ell, K) \bowtie W$ |

if originchk($L$) 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 reply($page, s, ck$) with $\vec{x} = \vec{s}$ 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 $\vec{s}$. Command redirect($\vec{u}, \vec{z}, ck$) with $\vec{x} = \vec{s}$ 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 models: 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 ($\perp$ when there is no active session).

The state of a server is modeled as a triple $(D, \phi, t)$ where the database $D$ is a partial map from names to memories, $\phi$ maps session identifiers (i.e., names) to the corresponding user identities, and $t$ is the parallel composition of multiple threads. Thread $u[\vec{r}](\vec{x}) \mapsto c$ 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 $[c]_E$ 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 $\ell$ 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 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 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 \( \tau \) annotated with their security type \( \tau \). 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 appendix A. In the following we write \( \lambda = \left( \pi, d, v \right) \) the browser model in appendix A. In the following we write \( \lambda = \left( \pi, d, v \right) \).

The labelling of cookies depends 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:

\[
(\text{http}(d) \land \text{https}(d), \ \lambda_{d, d'}(\text{http}(d') \land \text{https}(d')))
\]

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 \(^3\), hence the integrity is unchanged.\(^4\)

This behavior is represented by the following label:

\[
(\text{https}(d), \ \lambda_{d, d'}(\text{http}(d') \land \text{https}(d')))
\]

Cookie prefixes \(^5\) 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:

\[
(\text{https}(d), \ \lambda_{d, d'}(\text{http}(d')))
\]

The _Host– prefix strengthens the policy enforced by __Secure– 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:

\[
(\text{https}(d), \ \lambda_{d, d'}(\text{http}(d')))
\]

We discuss now the impact of HSTS \(^6\) on cookie labels. We use a set of domains \( \Delta \subseteq D \) to represent all the domains where HSTS is enabled, which essentially corresponds to the HSTS preload list \(^7\) that is shipped with modern browsers. Since

\(^2\) 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 \([33]\). For instance, www.example.com and atk.example.com are related domains, while example.co.uk and atk.co.uk are not.

\(^3\) Although most modern browsers forbid this dangerous practice, we have decided to represent the behavior dictated by the cookie specification.

\(^4\) https://hstspreload.org
HSTS prevents browsers from communicating with certain domains over HTTP; in practice it prevents network attackers from setting cookies by modifying HTTP responses coming from these domains. The label of a Secure cookie for domain $d$ becomes the following:

$$(\text{https}(d), \bigwedge_{d' \neq d} \text{http}(d') \land \bigwedge_{d' \neq d} \text{https}(d'))$$

The integrity label shows that the cookie can be set over HTTPS by any related domain of $d$ (as for Secure cookies) and over HTTP only by related domains where HSTS is not enabled. If HSTS is activated for $d$ and all its related domains, the cookie label becomes the same as that of cookies with the __Secure- prefix.

In the model we can also formalize attackers using labels which denote their read and write capabilities. Considering an attacker at label $\ell_a$ and a name with label $\ell$, the name may be learned by the attacker if $C(\ell) \subseteq C(\ell_a)$ and may be influenced by the attacker if $I(\ell_a) \subseteq I(\ell)$. Here we model the following popular attackers from the web security literature:

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:

$$\text{(http}(d) \lor \text{https}(d), \text{http}(d) \lor \text{https}(d))$$

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

$$\bigvee_{d \in D} \text{http}(d), \bigvee_{d \in D} \text{http}(d)$$

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:

$$\bigvee_{d' \neq d} (\text{http}(d') \lor \text{https}(d')),$$

$$\bigvee_{d' \neq d} (\text{http}(d') \lor \text{https}(d'))$$

D. Semantics

We present now the most relevant rules of semantics in [1] deferring to appendix [C] 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.

1) Servers: rules rely on the function $\text{eval}_E(se, D)$ that evaluates the expression $se$ in the environment $E$ using the database $D$. The formal definition is in appendix [C] here we provide an intuitive explanation. The evaluation of $@$ or $@r$ and $@r$ yields the value associated to $r$ in the global and the session memory identified by $E$, respectively. Expression $\text{fresh}(\ell)$ evaluates to a fresh name sampled from $N$ with security label $\ell$. 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 $\text{req}(\ell_b, n, u, p, ck, l)$. 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 $\vec{x}$ 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-RESTORE$\textit{SESSION}$) and (S-NEW$\textit{SESSION}$). If $se$ evaluates to a name $j \in \text{dom}(D)$, 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 $E = \vec{i}, \bot$ 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 $\rho$ mapping the pair $(\ell_s, u)$ to the correct password where $\ell_s$ is the identity of the user and $u$ is the login endpoint. The rule updates the trust mapping $\phi$ by associating the session identifier specified in the login command with the identity $\ell_s$.

Rules (S-OCHK$\textit{SUCCE}$) and (S-TCK$\textit{FAIL}$) 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. In case of a failure we produce the event error.

Rule (S-AUTH) produces the authenticated event $\vec{i}\vec{v}_{\ell_s}$ where $\vec{i}$ is data identifying the event, e.g., paper and action in the HotCRP example of Section III-B. The event is annotated with the identities $\ell_b, \ell_s$, representing the user running the browser and the account where the event occurred, and the label $\ell$ 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 $\text{send}$, The response contains a page $\vec{p}$, 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 $\vec{x}$. The third and the fourth component of $\text{send}$ 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 $\text{halt}$.

2) Web Systems: the semantics of web systems regulates the communications among browsers, servers and the attacker.

Rule (A-BRO$\textit{SER}$) synchronizes a browser sending a request $\text{send}$ 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 $C(\lambda(u)) \subseteq C(\ell)$.

Rule (A-BRO$\textit{ATK}$) 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
TABLE II: Semantics (excerpt).

**Servers**

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S-RECV)</td>
<td>( \alpha = \text{req}(\partial_b, n, u, p, c, k, l) ) ( R = n, u, b, l ) ( \forall k \in [1 \ldots m], v_k = (k \in \text{dom}(\pi)) ) ( \gamma ) ( \beta ) ( \sigma ) ( \forall j \in \mathbb{N} ) ( m =</td>
</tr>
<tr>
<td>(S-NEWSESSION)</td>
<td>( E = i, [\text{start } se]_k ) ( \text{eval}_g(se, D) = j ) ( j \notin \text{dom}(D) ) ( (D, \phi, [\text{start } se]_k)_k \mapsto (D \uplus {j \mapsto }) ), ( \phi ), ( [\text{skip}]_k )</td>
</tr>
<tr>
<td>(S-OCHKSUCCE)</td>
<td>( R = n, u, b, l ) ( \ell \in L ) ( (D, \phi, [\text{if originchk}(L) \text{ then } c]_k ) ( \mapsto (D, \phi, [c]_k) )</td>
</tr>
<tr>
<td>(S-AUTH)</td>
<td>( R = n, u, b, l ) ( j \in \text{dom}(\phi) ) ( \alpha = \bar{\alpha}<em>R[\phi[j]] ) ( \forall k \in [1 \ldots \gamma], \text{eval}</em>{i, j}(se, D) = v_k ) ( \gamma ) ( \beta ) ( \sigma ) ( \forall j \in \mathbb{N} ) ( m =</td>
</tr>
</tbody>
</table>

**Web systems**

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A-BROSER)</td>
<td>( W \triangleright {u, n, u, p, c, k, l} ) ( W' \triangleright {u, n, u, p, c, k, l} ) ( K' \triangleright (C(\lambda(u)) \subseteq C(\lambda)) ) ( \text{if } (K \cup \text{ns}(p, c, k)) : K ) ( \ell ) ( \triangleright ) ( \ell' ) ( K \triangleright (C(\lambda(u)) \subseteq C(\lambda)) ) ( \text{if } (K \cup \text{ns}(p, c, k)) : K ) ( \ell ) ( \triangleright ) ( \ell' )</td>
</tr>
<tr>
<td>(A-BROATK)</td>
<td>( W \triangleright (\ell, \kappa) ) ( K' \triangleright (\ell', \kappa') ) ( \triangleright ) ( W' )</td>
</tr>
<tr>
<td>(A-ATKSEP)</td>
<td>( n \in N ) ( n \neq \text{usr} ) ( \text{ns}(p, c, k) \subseteq K ) ( \text{if } \alpha = \text{req}(\partial_b, n, u, p, c, k, l) ) ( W \triangleright ) ( ) ( (\ell, \kappa) ) ( \triangleright ) ( (\ell, \kappa') \cup {n} ) ( \triangleright ) ( W' )</td>
</tr>
</tbody>
</table>

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 unattacked.

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

Traces include attacker actions, authenticated events \( \bar{\alpha}_R[\phi[j]] \) and \( \bullet \) denoting actions without visible effects or synchronizations not involving the attacker. Given a trace \( \gamma \), we write \( \gamma \downarrow (\ell, \ell') \) for the projection containing only the authentication events of the type \( \bar{\alpha}_R[\phi[j]] \) with \( j \in \{\ell, \ell'\} \). A trace \( \gamma \) is **unattacked** if it contains only \( \bullet \) actions, error events and authenticated events, otherwise \( \gamma \) is an **attacked** trace.

Now we introduce the definition of session integrity.

**Definition 2.** A web system \( W \) preserves session integrity against the attacker \( (\ell_a, K) \) for the honest user \( \text{usr} \) performing the actions \( \bar{\alpha} \) if for any attacked trace \( \gamma \) generated by the system \( (\ell_a, K) \triangleright B_{\text{usr}}(\{\}, \{\}, \emptyset) \| W \) there exists an unattacked trace \( \gamma' \) generated by the same system such that for all labels \( \ell \) we have:

\[ I(\ell_a) \nsubseteq I(\ell) \Rightarrow \gamma \downarrow (\text{usr}, \ell) = \gamma' \downarrow (\text{usr}, \ell). \]

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 **Appendix D** 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 \( T \), ranged
now we introduce the notion of $\ell$ parametrized by the attacker label $\ell_a$. Intuitively, security types inherit the subtyping relation for enforced by types.

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

\[
\ell \subseteq \ell' \\
\ell \subseteq_{\ell_a} \ell' \\
C(\tau) \sqsubseteq C(\tau') \subseteq C(\ell_a) \\
I(\ell_a) \subseteq I(\tau) \cap I(\tau')
\]

Intuitively, security types inherit the subtyping relation for labels but this is not lifted to the credentials, e.g., 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 $\Gamma = (\Gamma_U, \Gamma_X, \Gamma_{\mathcal{R}^a}, \Gamma_{\mathcal{R}^s}, \Gamma_V)$ is a 5-tuple and conveys the following information:

- $\Gamma_U : \mathcal{U} \rightarrow (\mathcal{L}^2 \times \mathcal{T} \times \mathcal{L})$ maps URLs to labels capturing the security of the network connection, the types of the URL parameters and the integrity label of the reply;
- $\Gamma_X : \mathcal{X} \rightarrow \mathcal{T}$ maps variables to types;
- $\Gamma_{\mathcal{R}^a}, \Gamma_{\mathcal{R}^s} : \mathcal{R} \rightarrow \mathcal{T}$ map global references and session references, respectively, to reference types;
- $\Gamma_V : \mathcal{V} \rightarrow (\mathcal{L}^2 \times \mathcal{T} \times \mathcal{L})$ 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 $\Gamma$ is well-formed for $\lambda$ and $\ell_a$ (written $\lambda, \ell_a, \Gamma \vdash \psi$) if the following conditions hold:

1) for all URLs $u \in \mathcal{U}$ with $\Gamma_U(u) = \ell_u, \tau , \ell$, we have:
   a) $C(\ell_u) = C(\lambda(u)) \land I(\lambda(u)) \subseteq I(\ell_u)$
   b) for all $k \in [1, \ldots, |\tau|]$ we have
      i) $C(\tau_k) \subseteq C(\ell_u) \land I(\ell_u) \subseteq I(\tau_k)$
      ii) $\tau_k = \text{cred}(\cdot) \land C(\tau_k) \subseteq C(\ell_u) \Rightarrow I(\ell_u) \subseteq I(\tau_k)$
2) for all references $r \in \mathcal{R}$ with $\Gamma_{\mathcal{R}^a}(r) = \tau$:
   a) $C(\tau) \subseteq C(\lambda(r)) \land I(\lambda(r)) \subseteq I(\tau)$
   b) for all $u \in \mathcal{U}$, if $C(\lambda(r)) \subseteq C(\lambda(u)) \land I(\ell_u) \subseteq I(\lambda(u))$ then $C(\tau) \subseteq C(\ell_u)$
   c) if $I(\ell_u) \subseteq I(\lambda(r))$ and $\tau = \text{cred}(\cdot) \land C(\tau) \subseteq C(\ell_u)$
   d) $\tau = \text{cred}(\cdot) \land C(\tau) \subseteq C(\ell_u) \Rightarrow I(\ell_u) \subseteq I(\tau)$

Conditions $[1a]$ and $[2a]$ ensure that the labels of URLs and cookies in the typing environment – which are used for the security analysis – are at most as strict as the labels in the function $\lambda$ introduced in Section III-C – which define the semantics. For instance, a cookie $r$ with confidentiality label $C(\lambda(r)) = \text{http}(d) \land \text{https}(d)$ is attached both to HTTP and HTTPS requests to domain $d$. It would be unsound to use a stronger label for typing, e.g., $\text{https}(d)$, 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 $[1b]$.

Conditions $[1b \text{ ii}]$ and $[2d]$ ensure that low confidentiality credentials – that can be learned and used by the attacker – cannot have high integrity.

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 $\ell_s$ 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, \( \ell_s \in L^2 \cup \{x\} \) where \( x \) denotes no active session.

The program counter label \( pc \in L \) 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 [39].

D. Explanation of the Typing Rules

1) Server Expressions: typing of server expressions is ruled by the judgement \( \Gamma, \ell_s \vdash_{\ell_k} se : \tau \), meaning that the expression \( se \) has type \( \tau \) in the typing environment \( \Gamma \) within the session \( \ell_s \). Names have type \( \text{cred}(\ell) \) where \( \ell \) is the label provided as an annotation (T-ENAME, T-EFRESH). Values different from names are constants of type \( \bot \), i.e., they have low confidentiality and high integrity (T-EVAL). Rule (T-EUNDEF) gives any type to the undefined value \( \bot \). 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-EGLOBAL). For session references we combine the information stored in the environment with the session label \( \ell_s \), which essentially acts as an upper bound on the types of references (T-SEESREF). In a honest session, \( \ell_s \) can have 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 and can never store secrets. 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 labels \( \ell_0 \times \ell_1 \). Values different from \( \bot \) have type \( \bot \) in the typing environment \( \Gamma \) within the session \( \ell_s \). 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-RESESREF) we have analogous conditions to (T-SEESREF) for session references. Subtyping for reference types is contra-variant to subtyping for security types (T-RSUB).

3) Server Commands: the judgement \( \Gamma, \ell_s, pc \vdash_{\ell_k, (u, b, P)} c : \ell_s', pc' \) states that the command \( c \) (bound to the endpoint at URL \( u \)) can be typed against the attacker \( \ell_s \) in the typing branch \( b \in \{\text{hon, csrf}\} \) using typing environment \( \Gamma \), session label \( \ell_s \) and program counter label \( pc, P \) 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 \( pc' \). We let \( C = (u, b, P) \) 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 (\( b = \text{hon} \)) or the CSRF case (\( b = \text{csrf} \)).

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 username, password and \( pc \) 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 \( (\bot, \ell_s) \) (since the attacker can access the session), otherwise we use the label \( \ell \) in the type of \( se \). Furthermore, we ensure that in the honest typing branch high integrity sessions can not be started or ended (by starting a new session) in a low integrity context (i.e., in a conditional with low integrity guard), since this can potentially influence the value of high integrity references of the session memory in the continuation. For the CSRF typing branch this is not required, since due to its low PC label it can never write to high integrity references.

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. If one of the branches contains a reply or a redirect command, then reaching the continuation depends on the taken branch, thus we use the join of the PC labels returned in the two branches to type-check the continuation; otherwise, we use the original PC label. 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
## Table III: Type system

### Server expressions

<table>
<thead>
<tr>
<th>(T-ENAME)</th>
<th>(T-EFRESH)</th>
<th>(T-EVAL)</th>
<th>(T-EUNDEF)</th>
<th>(T-EVAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{a} : \mathfrak{e}(l) )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{fresh}} \mathfrak{a} : \mathfrak{e}(l) )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{eval}} v \notin \mathcal{N} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{undef}} v : \bot )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{var}} x : \Gamma(x) )</td>
</tr>
</tbody>
</table>

### Server references

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{login}} s )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} s : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} C : \mathfrak{e}(l) )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} x : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
</tr>
</tbody>
</table>

### Server-side commands

<table>
<thead>
<tr>
<th>(T-IIf)</th>
<th>(T-SESSION)</th>
<th>(T-CHK)</th>
<th>(T-OKCHk)</th>
<th>(T-AUTH)</th>
<th>(T-REPLY)</th>
<th>(T-REDIR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} s : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} x : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{set}} \mathfrak{r} : \tau )</td>
</tr>
</tbody>
</table>

### Server-side commands

<table>
<thead>
<tr>
<th>(T-SUB)</th>
<th>(T-EBINOP)</th>
<th>(T-ESUB)</th>
<th>(T-ESUB)</th>
<th>(T-RGLOBRef)</th>
<th>(T-ESUB)</th>
<th>(T-EBINOP)</th>
<th>(T-SUB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{eval}} v \notin \mathcal{N} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{undefined}} v : \bot )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{var}} x : \Gamma(x) )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
<td>( \Gamma, \ell_s \vdash^\mathcal{E}_{\text{ref}} \mathfrak{r} )</td>
</tr>
</tbody>
</table>
TABLE III: Type system (continued).

Forms

(T-FORM)

\[ \Gamma_F(v) = \Gamma_F(u) = \ell_u, \bar{r}, \ell_r \quad I(\ell_u) \subseteq I(\ell_u) \quad p_c \subseteq I(\ell_u) \quad m = |\bar{r}| = |\ell_r| \quad \forall k \in [1 \ldots m], \Gamma, \ell_u \vdash p_c' z_k : \tau'_k \wedge \tau''_k \subseteq \tau_k \]

\[ \Gamma, v, p_c \vdash^I \Gamma_{\ell_u} \text{ form}(u, z) \]

Server threads

(T-RECV)

\[ \lambda, \ell_u, \Gamma^0(u) \vdash c \quad \Gamma^0(u) = \ell_u, \bar{r}, \ell_r \quad m = |\bar{r}| = |\ell_r| \quad \forall k \in [1 \ldots m], \Gamma, \ell_u \vdash p_c' z_k : \tau'_k \wedge \tau''_k \subseteq \tau_k \]

\[ \Gamma_X = x_1 : \tau_1, \ldots, x_m : \tau_m \quad \Gamma_X^0 = x_1 : (\bot, C), \ldots, x_m : (\bot, C) \quad \Gamma_X^0, \Gamma_X^0, \Gamma \vdash \ell_u \quad \Gamma_X^0, \Gamma_X^0, \Gamma \vdash \ell_u, (u, \text{bom}, z) \quad c : \odot I(\ell_u) \]

\[ \Gamma^0_{\ell_u}, p_c \vdash^I \ell_r \quad \ell_r \vdash^I \ell_r \]

\[ \Gamma^0_{\ell_u}, p_c \vdash^I u[r](\ell_r) \rightarrow c \]

label or any expression in \( \bar{r} \)) 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-PRUNECHK) 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 high integrity.

Rule (T-REPLY) combines the PC label with the expected integrity label of the response \( I_r \) for the current URL to compute \( p_c' \) which is used to type the response. In the honest typing branch, we require \( p_c' = \ell_r \), 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 \( p_c \) (cf. rule (T-BINCLUDE) in appendix [D]). Using the typing environment \( \Gamma' \) 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 appendix [D]);
- 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. Additionally, we require that the target URL is not relying on an origin check for CSRF protection \( (u \notin \mathcal{P}) \), as the redirect would allow for a circumvention of that protection. Finally, we also require that the expected integrity label for the response for the current URL and the target URL are the same.

4) Forms: the judgement \( \Gamma, v, p_c \vdash^I \ell_r \) says that a form \( f \) identified by the name \( v \) is well-typed in the environment \( \Gamma \) under the label \( p_c \). 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. We require that only links to high integrity URLs are included and with \( p_c \subseteq I(\ell_u) \) we check that the thread running with program counter label \( p_c \) is allowed to trigger requests to \( u \). In this way we can carry over the \( p_c \) from one thread where the form has been created to the one receiving the request since we type-check the honest branch with \( p_c = I(\ell_u) \). 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 \( \Gamma^0 \vdash^I \ell_r, \mathcal{P} \) says that the thread \( t \) is well-typed in the environment \( \Gamma^0 \) against the attacker \( \ell_r \) and \( \mathcal{P} \) is the set of URLs protected against CSRF
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., $t = [c] \frac{γ}{δ}$) are in appendix [10] since they are needed only for proofs.

Rule (T-RECV) checks that the environment is well-formed and that the network connection type $\ell_u$ 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 $b = \text{hon}$: parameters are typed according to the type of $u$ in $\Gamma^0_u$ which is reflected in the environment $\Gamma_X$. As the honest user initiated the request, we let $\rho_c = I(\ell_u)$, i.e., we use the integrity label of the network connection as $\rho_c$. 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 $b = \text{csrf}$: parameters are chosen by the attacker, hence they have type $(\bot, T_e)$ in $\Gamma_X$. As the attacker initiated the request, we let $\rho_c = T_f$.

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 appendix [10].

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 $\text{usr}$ in a web system $W$ with respect to a typing environment $\Gamma(u)$ and an attacker $\ell_a$ iff

1) for all actions $a'$ in $\vec{a}$ we have:

- if $a' = \text{load}(\text{tab}, u, p)$, $\Gamma_t(u) = \ell_u, \tau, l_r$ then for all $k \in \text{dom}(p)$ we have $p(k) = v' \Rightarrow \tau' \subseteq_{\ell_a} \tau_k$;
- if $a' = \text{submit}(\text{tab}, u, v', p)$, $\Gamma_v(v') = \ell_u, \tau, l_r$ then for all $k \in \text{dom}(p)$ we have if $p(k) = v''$ then $\tau' \subseteq_{\ell_a} \tau_k$. If $I(\ell_a) \subseteq_{\ell_a} \lambda(u)$ we additionally have $\tau' \subseteq_{\ell_a} \tau_a$.

2) $(\ell_a, K_a) \triangleright B_{\text{usr}}(\{\}, \{\}, \vec{a}) \parallel W \xrightarrow{\gamma} (\ell_a, K_a') \triangleright B_{\text{usr}}(M, P, \{\}) \parallel W'$ for some $K', W', M, P$ where $\gamma$ is an unattacked trace, not containing the event error;

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

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. For this reason we also require that the user never provides secrets to a form embedded in a page of low integrity.

Condition 2 lets us consider only honest runs in which the browser terminates regularly without producing errors. 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 $u[\rho][\vec{x}] \hookrightarrow c$. 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, $(\ell_a, K_a)$ an attacker, $\Gamma^0$ a typing environment, $P$ a set of protected URLs against CSRF via origin checking and let $\vec{a}$ be a list of well-formed user actions for $\text{usr}$ in $W$ with respect to $\Gamma^0$ and $\ell_a$. Assume that for all $u$ with $\rho(\text{usr}, u) = n^u$ we have $C(\ell) \subseteq_C C(\ell_a)$ and for all $n^u \in K$ we have $C(\ell) \subseteq_C C(\ell_a)$. Then $W$ preserves session integrity against $\ell_a$ with knowledge $K$ for the honest user $\text{usr}$ performing the list of actions $\vec{a}$ if $\Gamma^0 \vdash_{\ell_a, P} t$ 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 appendix [12].

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 appendix E.

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 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. 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 (from Section II-D) on domain dc against an attacker controlling a related-domain dcE ∼ dc, assuming that the session cookie is secured with the __Host- prefix. We let the attacker label le = (http(dCE) ∨ https(dCE), http(dC) ∨ https(dC)), and let lc = (https(dc), http(dc)), leLH = (lc, https(dc)), leHL = (https(dc), T_I). We then consider a minimal environment Γ sufficiently to type the login endpoint, where:

Γu = {login ⇒ (lc, (ℓLH, cred(ℓC), cred(ℓHL)), https(dC)), manage ⇒ (lc, (ℓC, ℓLH, cred(ℓHL)), https(dC))}
ΓR = {r ⇒ cred(ℓC), r' ⇒ cred(ℓHL), sid ⇒ cred(ℓC), pre ⇒ cred(ℓHL)}
ΓS = {user ⇒ ℓLH, token ⇒ cred(ℓHL)}
ΓV = {auth ⇒ Γu(login), link ⇒ Γu(manage)}

We type-check 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 https(dC). In the then branch (line 3), we have the assignment @r' := fresh{lH}, which types successfully according to (T-SETGLOBAL). The start statement with the freshly sampled value yields a session label le = (https(dC), T_I). The assignment $token := fresh{lH}$ also succeeds according to (T-SETSESSION). The session label does not affect the type of the reference $token$ in this case. For the reply (lines 4–6) 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

5 To simplify the presentation, 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.

6 Here we expose the annotations of fresh() expressions (needed for typing) that we omitted from Section II for readability purposes.
assigned to the cookie \textit{pre} (T-REPLY). In the \texttt{else} branch of the conditional, we start a session over the cookie \texttt{@pre} (line 8), leading to a session label \(\ell_s = (\text{https}(\text{dc}), \bot)\) (T-START). The conditions in (T-TCHK) are fulfilled for the \texttt{tokenchk} command (line 9) and we continue typing without any additional effect. Since we still have \(\texttt{pc} = \text{https}(\text{dc})\), the assignment \(\texttt{@r := fresh()^C}\) type-checks (line 10). As the password is of the same type as the reference \texttt{@r} containing the session secret, the \texttt{login} also type-checks successfully (T-LOGIN). The \texttt{start} statement over a credential of type \(\texttt{cred}(\ell_C)\) gives us the session label \(\ell_s = \ell_C\) (line 11). For the \texttt{reply} (lines 12–14), we check that we may include the form with the current \texttt{pc} and that it is well formed (trivial since it contains only \(\bot\)), that the empty script is well-typed and that we may assign the value of \texttt{@r} to the cookie \texttt{sid} (T-REPLY).

The \texttt{then} branch of the CSRF case types similarly to the honest case, since all references used in it and the cookie \textit{pre} have integrity label \(\top\). Additionally, in the CSRF branch, we do not type the DOM (T-REPLY). In the \texttt{else} branch we start a session (line 8) with label \(\ell_s = (\text{https}(\text{dc}), \top)\) (T-START). When performing the \texttt{tokenchk} (line 9), we can apply rule (T-PRUNETCHK), since \(\Gamma, \ell_s \vdash \bot\) \(\texttt{\$token : cred}(\ell_H)\) and \(\Gamma, \ell_s \vdash \ell_a\) \texttt{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 \cite{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 \cite{11} covers different research lines. We discuss here the papers which are closest to our work.

In the context of web sessions, \cite{12} employed reactive non-interference \cite{10} to formalize and prove strong confidentiality properties for session cookies protected with the \texttt{HttpOnly} and \texttt{Secure} attributes, a necessary condition for any reasonable notion of session integrity. A variant of reactive non-interference was also proposed in \cite{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 \cite{21}. A more general definition of web session integrity, which we adapted in the present paper, was introduced in \cite{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, \cite{14} proposed the adoption of micro-policies \cite{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 \cite{3} and \cite{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 \cite{1} and web-based cloud providers \cite{6}.

Web protocols for single sign-on have also been manually analyzed in the expressive Web Infrastructure Model (WIM): for instance, \cite{23} focused on OAuth 2.0, \cite{24} considered OpenID Connect, and \cite{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, \cite{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 SELink \cite{19}, UrFlow \cite{17}, SeLINQ \cite{37} and JSLINQ \cite{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 \cite{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 \cite{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


TABLE IV: Syntax of browsers.

<table>
<thead>
<tr>
<th>Browsers</th>
<th>Expressions</th>
<th>( \text{be} ) ::=</th>
<th>( x \mid \tau \mid v \mid \text{dom}(\text{be}, \text{be'}) \mid \text{be} \odot \text{be'} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scripts</td>
<td>( s ) ::=</td>
<td>\text{skip}</td>
<td>\text{setdom}(\text{be}, \text{be'}, u, \text{be})</td>
</tr>
<tr>
<td>Connections</td>
<td>( N ) ::=</td>
<td>{ } \mid { n \mapsto \langle \text{tab}, u, l \rangle }</td>
<td></td>
</tr>
<tr>
<td>Pages</td>
<td>( P ) ::=</td>
<td>{ } \mid { P \uplus { \text{tab} \mapsto \langle u, \text{page} \rangle } }</td>
<td></td>
</tr>
<tr>
<td>Tasks</td>
<td>( T ) ::=</td>
<td>{ } \mid { \text{tab} \mapsto s }</td>
<td></td>
</tr>
<tr>
<td>Output queue</td>
<td>( Q ) ::=</td>
<td>{ } \mid { \alpha }</td>
<td></td>
</tr>
<tr>
<td>Browsers</td>
<td>( B ) ::=</td>
<td>( \langle N, M, P, T, Q, \bar{a} \rangle )</td>
<td></td>
</tr>
</tbody>
</table>

APPENDIX

A. Browser Model

The syntax of the scripting language supported in our browser model is given in **Table IV**. We let \( \text{be} \) range over expressions including references (for cookies), values, DOM elements, and binary operations defined over expressions, e.g., arithmetic and logical operations. In particular, expression \( \text{dom}(\text{be}, \text{be'}) \) extracts a value from the DOM of the page where the script is running; the expression \( \text{be} \) identifies the tag of the form in the page, while \( \text{be'} \) specifies the parameter of interest in the form. For simplicity, we stipulate that \( \text{dom}(\text{be}, \text{be'}) \) selects the URL of the form if \( \text{be'} \) evaluates to 0.

Command **skip** does nothing, while \( s; s' \) denotes the standard command concatenation. Command \( r := \text{be} \) assigns to reference \( r \) the value obtained by evaluating the expression \( \text{be} \). Command **include** retrieves the script located at URL \( u \) providing \( \text{be} \) as parameters: we use this construct to model both contents inclusion and a simplified version of XHR requests which is not subject to SOP restrictions which are applied by real browsers. Command **setdom** substitutes a form in a page, where \( \text{be'} \) is the tag of the form to be replaced, \( u \) and \( \text{be} \) are respectively the URL and the parameters of the new form.

The state of a browser is \( \langle N, M, P, T, Q, \bar{a} \rangle \) where \( \bar{a} \) is the identity of the user who wants to perform the list of actions \( \bar{a} \). The network store \( N \) maps connection identifiers to triples \( (\text{tab}, u, l) \) where \( \text{tab} \) identifies the tab that initiated the connection, \( u \) is the contacted endpoint and \( l \) is the origin that has been sent in the Origin header of the request and it is needed to correctly handle the header during redirects. \( M \) is the cookie jar of the browser, which is modeled as a map from references to values. \( P \) maps tab identifiers to pairs \( (u, \text{page}) \) representing the URL and the contents of the web page and \( T \) tracks running scripts: if \( T = \{ \text{tab} \mapsto s \} \), script \( s \) is running on the page contained in \( \text{tab} \). Finally, \( Q \) is a queue (of maximum size 1) of browser requests that is needed to handle redirects in our model.

Finally, we presuppose the existence of the set of domains \( \Delta \subseteq D \) containing all domains where HSTS is enabled, which essentially models the HSTS preloaded list that is shipped with modern browsers.

When a cookie is set with a Domain attribute whose value is a domain \( d \), the cookie will be attached to all requests towards \( d \) and its subdomains. This behavior is modelled by the labelling

\[
\left( \bigwedge_{d \leq d'} \text{http}(d') \land \text{https}(d'), \bigwedge_{d' \sim d} (\text{http}(d') \land \text{https}(d')) \right)
\]

where \( \sim \) is a preorder defined on \( D \) such that \( d \leq d' \) iff \( d \) is subdomain of \( d' \).

We discuss now the impact of HSTS on cookie labels: since this security policy prevents browsers from communicating with certain domains over HTTP, essentially it prevents network attackers from setting cookies by modifying HTTP responses coming from these domains. In particular, the label for a Secure cookie for domain \( d \) becomes the following:

\[
\left( \text{https}(d), \bigwedge_{d' \sim d} \text{https}(d') \right)
\]

If HSTS is enabled for \( d \) and all its related domains, then the cookie label is the same as that of cookies with the **Secure** cookie prefix, i.e.:

\[
\left( \text{https}(d), \bigwedge_{d' \in \Delta} \text{https}(d') \right)
\]

C. Complete Semantics

1) Browsers: We present the browser semantics in **Table V** where we exclude non-deterministic behaviors by requiring that \( i \) at most one network connection is open at any time; \( ii \) the user performs an action only when there are no pending network connections and no script is running, which amounts to asking that the user waits that the current page is completely rendered. This design choice is made to simplify our security proof and it has no impact the expressiveness of our model.

First we define the semantics of expressions in terms of the function \( \text{eval}(be, M, f) \) that evaluates the expression \( be \) in terms of the cookie jar \( M \), the DOM of the webpage \( f \) and the security context \( \ell \). Rule (BE-REFERENCE) models the access to the cookie jar, which is allowed only if the confidentiality level of the reference is below that of the security context. Rule (BE-DOM) selects a value from the DOM of the page depending on the values of the expressions \( be \) and \( be' \). Rules (BE-VAL) and (BE-BINOP) are standard.

Our semantics relies on the auxiliary functions \( \text{get} _\text{ck} \) and \( \text{upd} _\text{ck} \) to select the cookies to be attached to an outgoing request and to update the cookie jar with the cookies provided in an incoming response, respectively. Given a cookie jar \( M \) and a URL \( u \), we let \( \text{get} _\text{ck}(M, u) \) be the map \( \text{ck} \) such that \( \text{ck}(r) = v \) iff \( M(r) = v \) and \( C(\lambda(r)) \subseteq C(\lambda(u)) \). Given a cookie jar \( M \), a URL \( u \) and a map of cookies \( \text{ck} \), we let \( \text{upd} _\text{ck}(M, u, \text{ck}) = M \uplus (ck \uplus u) \) where \( ck \uplus u \) is the map \( \text{ck'} \) such that \( \text{ck'}(r) = v \) iff \( \text{ck}(r) = v \) and \( I(\lambda(u)) \subseteq I(\lambda(r)) \).

We describe now the rules of the browser semantics. Rule (B-LOAD) models the loading of a new page as dictated by the action \( \text{load}(\text{tab}, u, p) \). The browser opens a new network connection represented by the fresh name \( n \) and sends a request to the server located at \( u \) providing the parameters
TABLE V: Semantics of browsers.

<table>
<thead>
<tr>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BE-VAL) $\text{eval}_B(b, M, f) = v$</td>
</tr>
<tr>
<td>$\text{eval}_B(v, M, f) = v$</td>
</tr>
</tbody>
</table>

(B-LOAD) $n \leftarrow N \quad \alpha = \text{eval}(t_b, n, u, p, ck, \bot) \quad (\text{orig}(u) = \text{http}(d) \Rightarrow d \notin \Delta) \quad (\{\}, M, P, \{\}, \{\}, \alpha)^{\|}$

(B-INCLUDE) $\forall k \in [1 \ldots |\mathcal{I}|]: p(k) = \text{eval}_{\mathcal{I}(u)}(b_k, M, f) \quad \alpha = \text{eval}(t_u, n, u, p, ck, \text{orig}(u')) \quad (\text{orig}(u') = \text{http}(d) \Rightarrow d \notin \Delta) \quad (\{\}, M, P, \{\}, \{\}, \alpha)^{\|}$

(B-RECVLOAD) $\alpha = \text{res}(n, u, \bot, ck, page, s) \quad \alpha' = (\text{page} = \text{error}) \quad (\text{halt} : \alpha' : \alpha) \quad (\{n \rightarrow (\text{tab, u, o}), M, P, \}, \{\}, \alpha)^{\|}$

(B-RECVINCLUDE) $\alpha = \text{res}(n, u, \bot, ck, page, s) \quad \alpha' = (\text{page} = \text{error}) \quad (\text{halt} : \alpha' : \alpha) \quad (\{n \rightarrow (\text{tab, u, o}), M, P, \}, \{\}, \alpha)^{\|}$

(B-DIRECT) $\forall k \in [1 \ldots |\mathcal{I}|]: p(k) = v_k \quad \alpha = \text{eval}(t_b, n, u, p, ck, \text{orig}(u)) \quad (\text{orig}(u') = \text{http}(d) \Rightarrow d \notin \Delta) \quad (\{n \rightarrow (\text{tab, u, o}), M, P, \}, \{\}, \alpha)^{\|}$

(B-SUBMIT) $\{\text{tab} \rightarrow (u, f)\} \in P \quad \forall k \in [1 \ldots |\mathcal{I}|]: p'(k) = k \in \text{dom}(p) \quad p(k) : v_k \quad n \leftarrow N \quad \alpha = \text{eval}(t_b, n, u, p, ck, \text{orig}(u)) \quad (\text{orig}(u') = \text{http}(d) \Rightarrow d \notin \Delta) \quad (\{n \rightarrow (\text{tab, u, o', v}), M, P, \}, \{\}, \alpha)^{\|}$

(B-FLUSH) $\{\text{tab} \rightarrow s\} \in P \quad (\{\}, M, P, \{\}, \alpha)^{\|}$

(B-SEQ) $\langle N, M, P, T, \{\alpha\}, \{\alpha\}\rangle \Rightarrow \langle N, M, P, T, \{\alpha\}\rangle$

(B-END) $\{\{\}, M, P, \{\}, \alpha \}^{\|}$

(B-SETREFERENCE) $\{\text{tab} \rightarrow (u, f)\} \in T \quad \ell = \lambda(u) \quad \text{eval}_B(b, M, f) = v \quad I(\ell) \subseteq I(\lambda(r)) \quad (\{\}, M, P, \{\}, \alpha)^{\|}$

(B-SETDOM) $\ell = \lambda(u') \quad \text{eval}_B(b, M, f) = v' \quad \forall k \in [1 \ldots |\mathcal{I}|]: v_k = \text{eval}_B(b_k, M, f) \quad (\{\}, M, P \uplus \{\text{tab} \rightarrow (u', f)\}, \{\}, \alpha)^{\|}$

(B-SKIP) $\{\{\}, M, P, \{\}, \alpha \}^{\|}$

(B-EXPR) $\{\{\}, M, P, \{\}, \alpha \}^{\|}$
p and attaching the cookies \(ck\) selected from the cookie jar, with an empty origin header, as represented by the action \(\text{recv}(\{t, n, u, p, ck, \bot\}).\) If the protocol of the URL \(u\) is HTTP, we only allow the request if HSTS is not activated for the domain. In the connections store we associate \(n\) to the triple \((tab, u, \bot)\). Similarly, rule (B-INCLUDE) models the embedding of a script with the \texttt{include} directive of our scripting language. Compared to (B-LOAD), the main differences are that \(i\) the list of expressions \(\vec{b}\) specified in the instruction are evaluated; \(ii\) the request contains the origin of the page where the script is executed. Notice that the execution of the script is paused until a response is received: this behavior is similar to what happens in standard browsers when embedding scripts or using synchronous XHR requests.

Rule (B-RECVLOAD) models the receiving of a webpage over a pending network connection, represented by the transition label \(\text{res}(n, u, \bot, \bot, ck, page, s)\). As a result, the connection \(n\) is closed, the cookie jar is updated with the cookies \(ck\) attached to the response, the content of the tab associated to \(n\) is replaced with the received page and the script \(s\) is executed in that tab. In case the page error is received, we prepend the action halt to the list of user actions: since this action is not be consumed by any of the semantic rules, this models a cautious user that interrupts the navigation when an unexpected error occurs during the navigation. Rule (B-RECVINCLUDE) is similar to the previous rule: the main differences are that \(i\) the page contained in \(tab\) is left unchanged and the one sent by the server is discarded, therefore the user continues interacting with the website even when the error page is received by the browser; \(ii\) the script \(s\) sent by the server is prepended to the script \(s'\) that is waiting to run on the page. Rule (B-REDIRECT) models the receiving of a redirect from the server to URL \(u'\) with parameters \(\vec{v}\), represented by the transition label \(\text{res}(n, u, u', \bot, \bot, ck, \bot, page, s)\). The cookie jar is updated with the cookies \(ck\) set in the response and a new request to \(u'\) with the appropriate cookies and parameters is prepared by the browser and added to the output queue. If the origin \(o\) of the original request matches the origin \(\text{orig}(u')\) of the new target, the origin header remains the same for the new request, otherwise it is set to \(\bot\). The redirect is only allowed if it respects the HSTS settings for the new target.

Rule (B-SUBMIT) models the user clicking on a link or submitting a form in the page identified by URL \(u\) which is currently open in the browser at the specified tab. For each parameter we first check if the user has inserted a value by inspecting the map \(p\), otherwise we fallback to the pre-filled parameter contained in the form. A new network connection is opened, cookies from the cookie jar are attached to the outgoing request and the HSTS settings are checked as in (B-LOAD). The origin of the request is the origin of the URL \(u\) of the page containing the form. Rule (B-FLUSH) outputs on the network the request in the output queue produced by rules (B-LOAD), (B-INCLUDE) (B-REDIRECT) and (B-SUBMIT).

The remaining rules describe how scripts are processed. Rule (B-SEQ) models sequencing of script commands, (B-SKIP) processes the \texttt{skip} command and (B-END) terminates the script execution. Rule (B-SETREFERENCE) models the setting of a cookie by a script, which is allowed if the integrity label of the reference is above that of the URL of the page where the script is running. Finally, rule (B-SETDOM) models the update of a form in the DOM of the page where the script is running.

2) Servers: In Table VI we give the rules of the server semantics that were not presented in Section III-D. Rule (S-SEQ) is used for sequencing commands, (S-SKIP) to evaluate \texttt{skip}, (S-IFTRUE) and (S-IFFALSE) for conditionals, (S-OCKFAIL) and (S-TCKSUC) cover the missing cases of origin and token check, (S-SETGLOBAL) and (S-SETSESSION) respectively update the reference of a value in the global memory and in the session memory. Rule (S-REDIRECT) models a redirect from the server to the URL \(u'\) with parameters \(\vec{z}\) that sets the cookies \(ck\) in the user’s browser. The page and script components of the action \(\text{res}\) are respectively the empty page and the empty script, as they will be anyway discarded by the browser. As in rule (S-REPLY) shown in Table II all occurrences of variables in \(\vec{z}\) contained in the response are replaced with the results of the evaluations of the corresponding expressions in \(\vec{v}\) and we stipulate that the execution terminates after sending the message. Finally, rules (S-LPARALLEL) and (S-RPARALLEL) handle the parallel composition of threads.

3) Web Systems: We report in Table VII the rules of the web systems semantics that were not presented in the body of the paper. Rules (W-LPARALLEL) and (W-RPARALLEL) model the parallel composition of web systems. Rule (A-NIL) is applied when no synchronizations between two entities occur.

Rule (A-SERBRO) models an honest server providing a response to a browser over a pending connection. Here the knowledge of the attacker is extended either if she can read the messages using her network capabilities. Rule (A-SERATK) models the reception of a response from an honest server by the attacker. We require that the attacker knows the connection identifier \(n\) to prevent her from intercepting arbitrary traffic and we extend her knowledge with the contents of the message. Rule (A-ATKBRO) models the attacker providing a response to a browser either using her network capabilities or a server under her control. In this case we require that the attacker is able to produce the contents of the response using her knowledge \(K\), which amounts to asking that all names in the response are known to the attacker.

Finally, rule (A-TIMEOUT) is used to process requests to endpoints not present in the system W (e.g., attacker-controlled endpoints in a run without the attacker): in such a case, we let the browser process an empty response.

D. Typing Rules for Scripts

Table VIII presents the typing rules that were not introduced in the body of the paper due to lack of space.

1) Browser Expressions: Typing of browser expressions is ruled by the judgement \(\Gamma, b \vdash_{\Pi_a} \sigma : \tau\), meaning that the expression \(\sigma\) has type \(\tau\) in the typing environment \(\Gamma\).
and typing branch $b$. Rules are similar to those for server expressions, but in this case we do not carry around the session label since there are no session references. Rule (T-BEDOM) is used to type reading data from the DOM, where we conservatively forbid reading from the DOM in the honest branch and use label $\ell_a$ otherwise, since we then know that the type of all values in the DOM is upper bounded by $\ell_a$.

2) **Browser References:** Typing of references in the browser is ruled by the judgment $\Gamma \vdash b \colon \text{ref}(r)$. Meaning that the reference $r$ has reference type $\text{ref}(\tau)$ in the environment $\Gamma$. Compared to server references, the main difference is that there are no session references on the browser side.

3) **Scripts:** The typing judgment for scripts $\Gamma, \rho \vdash b \colon \ell_a, \rho \colon s$ reads as follows: the script $s$ is well-typed in the environment $\Gamma$ under the program counter label $\rho$ in the typing branch $b$.

Three straightforward to type scripts are (T-BSKIP) that trivially does nothing, (T-SEQ) checks both the concatenated commands and (T-BASSIGN) handles reference assignments just like (T-SETGLOBAL).

In the honest branch, (T-BSETDOM) performs the same checks as (T-FORM), namely that the script with program counter label $\rho$ is allowed to trigger a request to URL $u$, that the parameters of the generated form respect the type of the URL, and that the type associated to the name of the form

---

**TABLE VI: Semantics of servers (remaining rules).**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SE-VAL)</td>
<td>$\text{eval}_E(v, D) = v$</td>
</tr>
<tr>
<td>(SE-BINOP)</td>
<td>$\text{eval}_E(se, D) = v \quad \text{eval}_E(se', D) = v'$</td>
</tr>
<tr>
<td>(SE-READGLOBAL)</td>
<td>$\text{eval}_{1,i}(\text{br}, D) = D(i, r)$</td>
</tr>
<tr>
<td>(SE-READSESSION)</td>
<td>$\text{eval}_{1,i}(\text{br}, D) = D(j, r)$</td>
</tr>
<tr>
<td>(SE-FRESH)</td>
<td>$n \leftarrow N \quad \text{eval}_{1,i}((\text{br}, D) = n)$</td>
</tr>
</tbody>
</table>

**Server**

<table>
<thead>
<tr>
<th>Rule Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S-SEQ)</td>
</tr>
<tr>
<td>(S-SKIP)</td>
</tr>
<tr>
<td>(S-IFTRUE)</td>
</tr>
<tr>
<td>(S-IFFALSE)</td>
</tr>
<tr>
<td>(S-OCHKFAIL)</td>
</tr>
<tr>
<td>(S-IFELSE)</td>
</tr>
<tr>
<td>(S-SE-RP)</td>
</tr>
<tr>
<td>(S-TPARALLEL)</td>
</tr>
<tr>
<td>(S-SETSESSION)</td>
</tr>
</tbody>
</table>

**TABLE VII: Semantics of web systems (remaining rules).**

<table>
<thead>
<tr>
<th>Rule Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(W-LPARALLEL)</td>
</tr>
<tr>
<td>(W-RPARALLEL)</td>
</tr>
<tr>
<td>(A-NIL)</td>
</tr>
<tr>
<td>(A-SERBRO)</td>
</tr>
<tr>
<td>(A-SERATK)</td>
</tr>
<tr>
<td>(A-ATKBRO)</td>
</tr>
</tbody>
</table>

---
matches the type of the URL. For the attacked case, we just require that all parameters have type \( \ell_a \), as in the CSRF branch in rule (T-REPLY). Notice that we restrict the first expression in setdom to be a value, so that we can statically look up the associated type in \( \Gamma_V \).

Rule (T-BINCLUDE) performs the same checks on the URL parameters as the previous rule, but additionally requires in the honest case that the integrity of the network connection is high to prevent an attacker from injecting her own script which would then be executed in the context of the original page. Furthermore, we require that the included URL is not protected by an origin check as otherwise an attacker could abuse this to indirectly trigger a CSRF with the expected origin. We also require that the expected integrity label of the reply of the included URL \( u \) is the same as the \( pc \) used to type the current script: this is needed since executing a script that was typed with a program counter label of higher integrity leads to a privilege escalation, e.g., it could write to a high integrity reference which the current script should not be allowed to do. Including a script of lower integrity is also problematic since we type scripts in the same context as the DOM of the page, thus we would allow a low integrity script to write into the current (high integrity) DOM.

### E. Formal Results

**Definition 6.** Let \( \vec{a} \) be a list of user actions containing \( a_k = \load(tab, u, p) \). The navigation flow initiated from \( a_k \) is the list of actions \( a_k :: \mathbf{nf}(\vec{a} \downarrow k, tab) \) where \( \vec{a} \downarrow k \) is the list obtained from \( \vec{a} \) by dropping the first \( k \) elements and function \( nf \) is defined by the following rules:

\[
\begin{align*}
\mathbf{nf}(\langle \rangle, tab) &= \langle \rangle \\
\mathbf{nf}(\load(tab, u, p) :: \vec{a}, tab) &= \vec{a}' \\
\mathbf{nf}(a :: \vec{a}, tab) &= a' \\
\mathbf{nf}(\vec{a}, tab) &= \vec{a}'
\end{align*}
\]

**F. Case Studies**

Besides the case study on HotCRP that we have presented in the body of the paper, we have also analyzed other two popular PHP applications: phpMyAdmin \[36\], a software for database administration, and Moodle \[34\], an e-learning platform. We discuss now the encoding of the session management logic in these applications and some session integrity vulnerabilities affecting them, either novel or taken from recent CVEs.

1) Moodle: we present now the \texttt{login} endpoint implementing the authentication logic on Moodle. The endpoint expects the cookie \( sid \) which is used to store session data and the credentials of the user, namely the username \( uid \) and the password \( pwd \). Its encoding in our calculus is the following:

1. \( \texttt{login}[sid]([uid, pwd]) \leftarrow \)
2. \( \texttt{if} @\texttt{sid} = @\texttt{\_\_\_\_} \texttt{then} \)
3. \( \texttt{@\texttt{sid}} = \texttt{fresh}(); \)
4. \( \texttt{start} @\texttt{sid}; \)
5. \( \texttt{if} @\texttt{sid} @\texttt{\_\_\_\_} \texttt{then} \)
6. \( \texttt{redirect} (\texttt{profile}, \{\}), []) \)
7. \( \texttt{else} if \texttt{sid} = @\texttt{\_\_\_\_} \texttt{then} \)
8. \( \texttt{reply} ([\texttt{auth} \rightarrow \texttt{form}(\texttt{login}, \{\texttt{\_\_\_\_}, \texttt{\_\_\_\_}\}), \texttt{skip}, \{\texttt{sid} \rightarrow x\}) \)
9. \( \texttt{with} x = @\texttt{sid}; \)
10. \( \texttt{else} \)
11. \( \texttt{@\texttt{sid}} = \texttt{fresh}(); \texttt{login} \texttt{uid, pwd, @\texttt{sid}; start} @\texttt{sid}; \)
12. \( \texttt{if uid = uid; $sesskey = fresh}(); \)
13. \( \texttt{redirect} (\texttt{profile}, \{\texttt{sid} \rightarrow x\}) \) with \( x = @\texttt{sid}; \)

### Table VIII: Typing rules for scripts.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-BEVAR)</td>
<td>( \Gamma, b \vdash_{\ell_a} x : \Gamma_X(x) )</td>
<td>( \Gamma, b \vdash_{\ell_a} r : \Gamma_X(r) )</td>
</tr>
<tr>
<td>(T-BEREF)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T-BEVAL)</td>
<td>( v \notin N )</td>
<td>( \Gamma, b \vdash_{\ell_a} v : \bot )</td>
</tr>
<tr>
<td>(T-BUndef)</td>
<td></td>
<td>( \Gamma, b \vdash_{\ell_a} \bot : \bot )</td>
</tr>
<tr>
<td>(T-BNAME)</td>
<td></td>
<td>( \Gamma, b \vdash_{\ell_a} \text{cred}(\ell) )</td>
</tr>
<tr>
<td>(T-BEDOM)</td>
<td>( b \neq \text{hon} )</td>
<td>( \Gamma, b \vdash_{\ell_a} \text{dom}(be, be') : \ell_a )</td>
</tr>
<tr>
<td>(T-BEOPN)</td>
<td>( \Gamma, b \vdash_{\ell_a} s : \tau ) ( \text{and} ) ( \Gamma, b \vdash_{\ell_a} s' : \tau' )</td>
<td>( \tau = \ell \land \tau = \ell' ) ( v \land i ) = ( \vec{a} )</td>
</tr>
<tr>
<td>(T-BESUB)</td>
<td>( \Gamma, b \vdash_{\ell_a} b : \tau' ) ( \tau' \subseteq \ell_a \tau )</td>
<td>( \Gamma, b \vdash_{\ell_a} \bot : \tau )</td>
</tr>
<tr>
<td>(T-BREF)</td>
<td></td>
<td>( \Gamma, b \vdash_{\ell_a} r : \ref(\tau') ) ( \tau' \subseteq \ell_a \tau' )</td>
</tr>
<tr>
<td>(T-BASSIGN)</td>
<td>( \Gamma, b \vdash_{\ell_a} r : \ref(\tau) )</td>
<td>( \Gamma, b \vdash_{\ell_a} b : \tau ) ( \text{pc} \subseteq I(\tau) )</td>
</tr>
<tr>
<td>(T-BSKIP)</td>
<td></td>
<td>( \Gamma, b \vdash_{\ell_a} \text{skip} )</td>
</tr>
<tr>
<td>(T-BSETDOM)</td>
<td>( \Gamma, b \vdash_{\ell_a, p} \text{setdom}(v, u, be) )</td>
<td>( b = \text{hon} \Rightarrow \Gamma_Y(v) = \Gamma_Y(u) \land \text{pc} \subseteq I(\ell_a) \land \forall k \in [1 \ldots m], \tau_k \subseteq \ell_a \tau_k )</td>
</tr>
<tr>
<td>(T-BINCLUDE)</td>
<td>( \Gamma, b \vdash_{\ell_a, p} \text{include}(v, u, be) )</td>
<td>( b = \text{hon} \Rightarrow \forall k \in [1 \ldots m], \tau_k \subseteq \ell_a \tau_k \land u \notin \mathcal{P} )</td>
</tr>
</tbody>
</table>

**Table VIII**: Typing rules for scripts.
If no cookie $sid$ has been provided, e.g., when the user visits the website for the first time, a fresh cookie is generated (lines 2–3). The session identified by $sid$ is then started (line 4): if the identifier denotes a valid session, session variables stored when processing previous requests are restored. If the user previously authenticated on the website, the session variable $\$uid$ is different from the undefined value $\bot$ and a redirect to the profile endpoint (that here we do not model) is sent to the browser (lines 5–6). If the user is not authenticated and did not provide a pair of credentials, the server replies with a page containing the login form and a new cookie $sid$ is set into the user’s browser (lines 7–9). Finally, if the user has provided valid credentials, the endpoint starts a fresh session (to prevent fixation), stores in the session memory the user’s identity and a fresh value in $\$sesskey$ which is used to implement CSRF protection, then redirects the user to the profile endpoint and sets the new session identifier in the cookie $sid$ in the user’s browser (lines 10–13).

Since login does not perform any origin or token check before performing the login command, the endpoint is vulnerable to Login CSRF attacks, as it was the case for Moodle until November 2018 \[31\]. As discussed in Section II-D for HotCRP, this problem is captured when typing since the cookie must be of low integrity since no CSRF check is performed when it is set, therefore it cannot be used to perform authenticated actions of high integrity.

The solution implemented by Moodle developers uses pre-sessions, as we proposed for HotCRP in Section II In particular, developers decided for convenience to use the same cookie to handle both pre-sessions and sessions: this promotion of the cookie from low integrity, to handle the pre-session, to high integrity, when the session identifier is refreshed after authentication, cannot be modeled in our type system since we have a single static type environment for references, therefore type-checking would fail. In our encoding we model the fix by using two different variables for each input of interest, e.g., $g_{pwd}$ and $p_{pwd}$ for the password when provided via GET or POST, respectively.

2) phpMyAdmin: we show now the encoding of the session management logic for phpMyAdmin. In the following we model two HTTPS endpoints hosted on domain $d_P$: login, where database administrators can authenticate using their access credentials, and drop, where administrators can remove databases from the system.

We briefly discuss some implementation details of phpMyAdmin before presenting our encoding of the endpoints:

- for CSRF and login CSRF protection, phpMyAdmin inspects all incoming POST requests to check whether they contain a parameter token which is equal to the value stored in the (pre-)session memory;
- the parameters provided by the user are retrieved using the \$_REQUEST array which allows to uniformly access POST and GET parameters: in our encodings we model this behavior by using two different variables for each input of interest, e.g., $g_{pwd}$ and $p_{pwd}$ for the password when provided via GET or POST, respectively;
- a single cookie is used for pre-sessions and sessions while, as in the case of Moodle, we use two cookies $pre$ and $sid$;
- upon authentication, username and password are stored encrypted in two cookies: in our model we store them in the clear and use strong cookie labels to provide cookies with the confidentiality and integrity guarantees given by encryption.

We start with the encoding of the login endpoint. As parameters it expects the username and the password, both provided via GET and POST, and the login CSRF token, while as cookies we have $pre$ for the pre-session, $uid$ and $pwd$ where the credentials are stored upon authentication. The encoding in our calculus is the following:

```
1. login[pre](uid, pwd, ltoken) →
2. if $\pre = \bot$ then
3. $\pre = fresh();$
4. start $\pre;$
5. if $\uid \neq \bot$ then
6. redirect (profile, {});
7. else if uid = $\bot$ then
8. if $\token = \bot$ then
9. $\token = fresh();$
10. reply ($\{auth → form(login, (\bot, \bot, x))\}$, skip, $\{pre → y\}$)
11. with $x = \token$, $y = \pre$
12. else
13. $\token = \token$; $\token = fresh();$
14. if $\tokenchk(\token)$ then
15. $\token = fresh();$ login uid, pwd, $\sid$; start $\sid$
16. $\sid = uid$; $\$sesskey = fresh();$
17. redirect (profile, $\{sid → x, pre → y\}$)
18. with $x = \sid$, $y = \sid$
```

The main differences compared to the previous encoding are the following: i) the endpoint now expects a third $ltoken$ which is used to implement CSRF protection (line 1); ii) the login form is enriched with a CSRF token which is stored in the pre-session memory (lines 8–11); iii) the token stored in the session memory is compared to the one provided by the user before performing the authentication (line 14). After applying the fix, it is possible to perform high integrity authenticated actions within session started from the cookie $sid$ since it is possible to assign it a high integrity credential type when type-checking against the web attacker.
First the endpoint checks whether the user is already authenticated by checking whether cookies $uid$ and $pwd$ are provided: in this case, the user is redirected to the index endpoint (that here we do not model) showing all the databases available on the website (lines 2–3). Next the session identified by cookie $pre$ is started or a fresh one is created (lines 4–6). If the user has not sent her credentials, the page replies with a page containing the login form. This form contains a fresh CSRF token that is randomly generated for each request and stored in the session variable $token$. The response sent by the server contains the fresh pre-session cookie generated by the server (lines 7–10). Finally authentication is performed: a fresh session is started, a new token for CSRF protection is generated and the user is redirected to the index endpoint. The response sets into the user’s browser the cookies for session management and those containing the credentials. The only difference is that when login is performed via POST then the token checking is performed (lines 11–16), otherwise it is not (lines 17–21).

Now we present the encoding for the drop endpoint, where we let $\ell = (\text{https}(d), \text{https}(d))$. The endpoint expects three cookies: the session cookie $sid$ and those containing the credentials stored during the login. As parameters, it expects the name of the database to be deleted (provided either via GET and POST) and the CSRF token. The encoding in our calculus follows:

```
1. drop[sid, uid, pwd](g_db, p_db, token) \to
2. if $@uid = \bot$ or $@pwd = \bot$ then
3. redirect (login, {}, {});
4. start $@sid$;
5. if $p_db \neq \bot$ then
6. if tokenchk(token, $token$) then
7. auth $@uid, @pwd, g_db$ at $\ell$;
8. else
9. auth $@uid, @pwd, g_db$ at $\ell$;
10. reply ({}, skip, {});
```

First the endpoint checks where the user is authenticated by inspecting the provided cookies: if it is not the case, the user is redirected to the login endpoint (lines 2–3). After starting the session identified by the cookie $sid$, the endpoint drops the specified database after authenticating to the DBMS using the credentials stored in the cookies: this operation is abstractly represented using the auth command. Like in the login endpoint, the CSRF token is verified when the database to be removed is provided via POST (lines 5–7) and not if sent via GET (lines 8–9).

Both endpoints are vulnerable to CSRF attacks due to the security-sensitive commands performed without any token or origin check: the login command in login on line 18 and the auth command in drop on line 9. Until December 2018, several sensitive endpoints of phpMyAdmin where vulnerable to CSRF vulnerabilities analogous to the one presented for the drop endpoint [30]. [32]. The login CSRF, instead, is a novel vulnerability that we have discovered and has been recently assigned a CVE [33].

Type-checking captures the issue for the login CSRF vulnerability for the same reason of the other case studies, namely that the session cookie must be typed as low integrity and this prevents performing high integrity actions in the session. The standard CSRF is captured since it is not possible to apply rule (T-AUTH) when typing the auth of the drop endpoint in the csf typing branch.

The fix implemented by phpMyAdmin developers is the same for both vulnerabilities, i.e., using the $_REQUEST array rather than the $_POST array to retrieve the parameters provided by the user: this ensures that all sensitive operations are performed via POST, thus the CSRF token is always checked. To model this fix in our encoding we just get read of the input variables that represent GET parameters and remove the authenticated actions involving them. The encoding of the login endpoint becomes the following:

```
1. login[pre, uid, pwd](p_uid, p_pwd, token) \to
2. if $@uid \neq \bot$ and $@pwd \neq \bot$ then
3. redirect (index, {}, {});
4. if $@pre = \bot$ then
5. $@pre = \text{fresh}()$;
6. start $@pre$;
7. if $p_uid = \bot$ then
8. $token = \text{fresh}()$;
9. reply ({auth \to form(login, {\bot, \bot, x})}, skip, {pre \to y})
10. with $x = token, y = @pre$;
11. else if tokenchk(token, $token$) then
12. $@sid = \text{fresh}();$ login $p_uid, p_pwd, @sid$;
13. start $@sid$; $token = \text{fresh}()$;
14. redirect (index, {}, {uid \to x, pwd \to y, pre \to z, sid \to z})
15. with $x = p_uid, y = p_pwd, z = @sid$;
```

The encoding of the fixed drop endpoint is the following:

```
1. drop[sid, uid, pwd](g_db, token) \to
2. if $@uid = \bot$ or $@pwd = \bot$ then
3. redirect (login, {}, {});
4. start $@sid$;
5. if tokenchk(token, $token$) then
6. auth $@uid, @pwd, g_db$ at $\ell$;
7. reply ({}, skip, {});
```

After applying the fix, it is possible to successfully type-check our encoding of the phpMyAdmin session management logic against the web attacker.
In this section we present the full formal proof for the main result of the paper. The proof consists of two major parts: Subject Reduction ensures that typing and other invariants are preserved during execution of a web system. A relational invariant ensures that the attacked system and the unattacked system

G. Outline

In appendix [3] we introduce notation and helper functions used in the proof.

In appendix [4] we present an extended version of the semantics, containing additional annotations, as well as additional or modified typing rules needed to type running code. We show that the semantic rules are equivalent to the ones presented in the paper and that typing with the original rules implies typing with the extended typing rules.

In appendix [5] we prove the property of subject reduction for the system: This tells us that all components of the system are well-typed and that certain invariants are preserved during the execution of a single system.

In appendix [6] we introduce a relation between two websystems, that intuitively captures their equality on all high integrity components. We show that an attacked websystem is always in relation with its unattacked version and that this relation is preserved under execution.

In appendix [7] we combine results from the previous sections to show our main theorem.

H. Preliminaries

Here we introduce some notation that will be used in the remainder of the proof

Definition 7 (Notation). We define the following functions:

- For a websystem W we define servers(W) to be the set of all servers in W. For a websystem with attacker A = (\ell_a, \mathcal{K}) \triangleright W we let servers(A) = servers(W)
- For a websystem W we define browsers(W) to be the set of all browsers in W. For a websystem with attacker A = (\ell_a, \mathcal{K}) \triangleright W we let browsers(A) = browsers(W)
- For a server S = (D, t, \phi) we define urls(S) to be the set of all threads in t of the form \text{url}^{\text{path}}(\vec{x}) \rightarrow c.
- For a server S = (D, t, \phi) we define running(S) to be the set of all threads in t of the form \text{l}_n^{\mu, E}.
- For a thread of the form t = \text{l}_n^{\mu, E} we let \text{int}(t) = l.
- For a thread of the form t = \text{l}_n^{\mu, E} and a database of global memories D@ we let \text{mem}_p(D@, t) = D@.(i).
- For a thread of the form t = \text{l}_n^{\mu, E} and a database of session memories D_S we let \text{mem}_s(D_S, t) = D_S.(j).
- For a reference type \tau_r = \text{ref}_\ell(\tau) we let \text{ref}_\ell(\tau_r) = \tau.
- For a command c we let \text{com}(c) be the set containing all commands in c.
- For an event \alpha \in \mathcal{K} we define syncI(\alpha \in \mathcal{K}) = l as the sync integrity of the event.
- We define a meet \sqcap between a type \tau and a label \ell that limits the label of \tau to the label \ell. Formally:

\[\ell \sqcap \ell' = \begin{cases} \ell' \cap \ell & \text{if } \tau = \ell' \\ \text{cred}(\ell' \cap \ell) & \text{if } \tau = \text{cred}(\ell') \end{cases}\]

- We define a join \sqcup on types that behaves like the regular join \tau_1 \sqcup \tau_2 if it is defined and label(\tau_1) \sqcup label(\tau_2) otherwise.
- We define a join \sqcup_I as \text{cred}(\ell) \sqcup_I \ell := \text{cred}(((C(\tau), I(\tau) \sqcup_I \ell)) and \text{int}_I(l) := (C(\tau), I(\tau) \sqcup_I \ell)
- For a running server thread t = [c]^{\mu, E}_n we let \text{int}_I(t) = l \sqcup_I \bigcup_{t' \in \mathcal{E}} \text{reset}(t' \in c) I'
- For value \nu^c, we define jlabel(\nu^c) = (C(\tau) \sqsubseteq_C C(\ell_a)) : (\bot_C, \top_I) ? label(\tau).
- For a typing environment \Gamma and two memories M and M', we write M =_{\Gamma, \ell} M' if for all r with I(\ell_a) \sqsubseteq_I I(\text{ref}_r(\Gamma(r))) we have M(r) = M'(r).
- For a set of name \mathcal{N}, we let \mathcal{N}|_{\ell_a} be the set same set of names, where all types have been lowered to \ell_a.

Definition 8 (Freshness). \bullet A Browser B = (N, K, P, T, Q, \alpha)^{\text{pb}} is fresh if N = \{\}, K = \{\}, P = \{\}, T = \{\}, Q = \{\}.
- A Server S = (D, \Phi, t) is fresh if D = \{\} and \Phi = \{\}. (also see definition [3])
- A Websystem W is fresh if all B \in browsers(W) and all S \in servers(W) are fresh.

I. Extended Semantics and Typing Rules

In this section we introduce additional and modified rules for the semantics and the type system.

The most important changes are presented here:

- We annotate running server threads, the browser state, the DOM, and network requests and replies with an integrity label \ell \in \mathcal{L} and an attacked state \mu \in \{\text{hon}, \text{att}\}. Intuitively, \ell is dynamically tracking which domains have influenced the current state of the execution, while \mu is a binary flag that tells us whether the attacker used his capabilities to directly influence the current state.
• We annotate events with an integrity label (an additional one, using the notation $\alpha @ l$). This label is used to synchronize the execution of the unattacked and the attacked websystem in the relation: High integrity events have to be processed in sync, while low integrity events may be processed individually.

• We introduce a new command reset $l$ for servers to "reset the pc" after a conditional. This operation has no semantic effect, it just updates the integrity annotation.

• We partition the database $D = (D_0, D_1)$ into two different mappings for global and server memories.

• We split the rule (A-TIMEOUT) into two separate rules (A-TIMEOUTSEND) and (A-TIMEOUTRECV). We therefore introduce a buffer in the network state that keeps track of open connections that require a response. This is required since in the relation proof, every request and response needs to be atomic, so that it can be matched with the corresponding request or response in the other system. For example a request to a low integrity domain, that is intercepted by the attacker might be processed using a timeout in the unattacked system.

• All values $v^\tau \in V$ (in the code, in the DOM, in memory or in requests and responses) are now annotated with a security type that gives us runtime information. All primitive values have by default the type $\tau = \bot$ and hence can be given any security label $\ell$ (due to subtyping). Since for names $n^\tau \in N$ we have $\tau = cred(\ell)$ for some $\ell$, we cannot use subtyping if $C(\tau) \ntriangleright C(\ell)$ or $I(\ell) \ntriangleright I(\tau)$. We hence partition the set of names $N = N_0 \cup C(\ell) \cup C(\ell) \cup I(\ell, l) \cup I(\ell) \cup I(\ell, l, N_0)$ into one set $N_0$ of names of low confidentiality and integrity and one set $N_l$ for each label $\ell$ with high confidentiality or integrity.

We define a translation $\tau$ function from a fresh web system in the original semantics to websystems in the extended semantics. Intuitively, the translation annotates all constants with the type $\bot$ and lets the initial browser start with high integrity and in the honest mode.

\[
\begin{align*}
(l_a, \mathcal{K}) \triangleright W &\quad=\quad (l_a, \mathcal{K}) \triangleright W \\
W \parallel W' &\quad=\quad W \parallel W'' \\
\text{skip} &\quad=\quad \text{skip} \\
\text{auth} se at l &\quad=\quad \text{auth} se at l \\
\text{if } tokenchk(e, e') \text{ then } c &\quad=\quad \text{if } tokenchk(\ell, \ell') \text{ then } \ell \\
\text{if } originchk(L) \text{ then } c &\quad=\quad \text{if } originchk(L) \text{ then } \ell \\
\text{reply } (page, s, ck) \text{ with } \vec{x} = se &\quad=\quad \text{reply } (\vec{page}, \vec{s}, ck) \text{ with } \vec{x} = se \\
\text{redirect } (u, \vec{x}, ck) \text{ with } \vec{x} = se &\quad=\quad \text{redirect } (\vec{u}, \vec{x}, ck) \text{ with } \vec{x} = se \\
\text{fresh}(\ell) &\quad=\quad fresh(\ell') \\
se \odot se' &\quad=\quad se \odot se' \\
\ell &\quad=\quad \ell^{cred(\ell)} \\
\ell' &\quad=\quad \ell^{cred(\ell)} \\
\text{with } \ell \notin N \\
\{\}, \{\}, \{\}, \{\}, \{\}, \vec{a}\rangle_{l, \ell, hon} &\quad=\quad (\{\}, \{\}, \{\}, \{\}, \{\}, \vec{a}\rangle_{l, \ell, hon})
\end{align*}
\]

The extended semantics are presented in table IX, table XI, and table XII. As a convention, we use $\stackrel{\alpha}{\Rightarrow}$ for steps derived using the extended semantics and $\stackrel{\alpha}{\Rightarrow}$ for steps derived using the original semantics.

1) Detailed explanation of changes to browser semantics:

• The definition of $upd\_ck\!():\!(\cdot, \cdot, \cdot)$ is like the original definition of $upd\_ck\!():\!(\cdot, \cdot, \cdot)$, with the difference that the type annotations of values are joined with the type of the reference in the environment $\Gamma$. We will show in the proof that typing then ensures that the types of values in a memory reference is always equal to the type for that reference in the environment $\Gamma$.

• (BE-VAL) simply adds the value type
TABLE IX: Extended semantics of browsers.

$upd:\ck'((M, u, \ck)) = M \triangleright (\ck \triangleright' u)$ where $\ck \triangleright' u$ is the map $\ck'$ such that $\ck'(r) = v^\triangleright$ and $I(\lambda(u)) \subseteq I(\lambda(r))$.

Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>(BE-VAL)</th>
<th>(BE-BINOP)</th>
<th>(BE-REFERENCE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\eval(f, M, f) = v^\triangleright$</td>
<td>$\eval(f, M, f) = v^\triangleright'$</td>
<td>$\eval(f, \triangleright, \triangleright')$</td>
<td>$\eval(f, M, f) = M(\triangleright)$</td>
</tr>
</tbody>
</table>

(B-DOM)

$\eval(\triangleright, M, f) = v^\triangleright$ $\eval(\triangleright', M, f) = v^\triangleright'$ $\{v \mapsto \form(u^\triangleright, v^\triangleright')\} \subseteq f$ $\Rightarrow v = v^\triangleright$ $\Rightarrow v^\triangleright' = v^\triangleright'$

<table>
<thead>
<tr>
<th>Expression</th>
<th>(B-LOAD)</th>
<th>(B-INCLUDE)</th>
<th>(B-SUBMIT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha = \text{res}(i_b, n, u, p, \ck, \triangleright) \uplus I(\lambda(u), \triangleright', \text{hon})$</td>
<td>$\forall k \in [1, \ldots, #\triangleright\times #p]: p(k) = \eval(\lambda(u)(i_b), M, f)$</td>
<td>$\forall k \in [1, \ldots, #\triangleright\times #p]: (p(k) = k \in \dom(p)) \Rightarrow p(k) \Rightarrow n \leftarrow N$</td>
<td>${{}, M, P, {}, \text{submit}(t, u, v, p)} : \alpha \uparrow, \triangleright', \text{hon}$</td>
</tr>
</tbody>
</table>

(B-RECVLOAD)

$\alpha = \text{res}(i_b, n, u, \triangleright, \{\}, \ck, p, \text{orig}(u)) \uplus I(\lambda(u), \mu', \triangleright')$ $\Rightarrow \ck' = \text{get}\ck(M, u)$ $\Rightarrow \{\{\}, M, P, \{\}, n \rightarrow (\triangleright, \text{orig}(u))\}$ $\Rightarrow \ck' = \text{get}\ck(M, u)$

(B-RECVINCLUDE)

$\alpha = \text{res}(i_b, n, u, \triangleright, \{\}, \ck, p, \text{orig}(u))$ $\Rightarrow \ck' = \text{get}\ck(M, u)$ $\Rightarrow \{\{\}, M, P, \{\}, n \rightarrow (\triangleright, \text{orig}(u))\}$ $\Rightarrow \ck' = \text{get}\ck(M, u)$

(B-REDIRECT)

$\ck' = \text{get}\ck(M, u)$ $\Rightarrow \{\{\}, M, P, \{\}, n \rightarrow (\triangleright, \text{orig}(u))\}$ $\Rightarrow \ck' = \text{get}\ck(M, u)$

(B-FLUSH)

$\{\{\}, M, P, \{\}, \text{skip}\} : \alpha \uparrow, \triangleright, \text{hon}$

(B-SEQ)

$\{\{\}, M, P, \{\}, \text{skip}\} : \alpha \uparrow, \triangleright, \text{hon}$

(B-SETREFERENCE)

$\{\{\}, M, P, \{\}, \text{skip}\} : \alpha \uparrow, \triangleright, \text{hon}$

(B-SETDOM)

$\eval(\triangleright, M, f) = v^\triangleright$ $\forall k \in \[1, \ldots, \#\triangleright\times \#p\]: \ck'' = \eval(\triangleright, M, f) \triangleright u^\triangleright' \Rightarrow \mu'' = (\mu = \text{att} \lor \mu' = \text{att}) \lor \text{hon}$

$\ck' = \text{get}\ck(M, u)$ $\Rightarrow \ck' = \text{get}\ck(M, u)$

(B-END)

$\{\{\}, M, P, \{\}, \text{skip}\} : \alpha \uparrow, \triangleright, \text{hon}$
### Table X: Extended semantics of server expressions.

<table>
<thead>
<tr>
<th>Expressions</th>
<th>(SE-VAL)</th>
<th>(SE-FRESH)</th>
<th>(SE-BinOp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval(_E\left(v^r, D\right) = \nu^r)</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{usr}^r\right))</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{not,usr}^r\right))</td>
<td></td>
</tr>
<tr>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{usr}^r\right))</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{not,usr}^r\right))</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{not,usr}^r\right))</td>
<td></td>
</tr>
<tr>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{usr}^r\right))</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{not,usr}^r\right))</td>
<td>eval(<em>E\left(\nu^r \leftarrow \mathcal{N}</em>\text{not,usr}^r\right))</td>
<td></td>
</tr>
</tbody>
</table>

- (BE-BinOp) adds types, and assigns the join of the labels of the input types to the result.
- (BE-Reference) is the same as in the original semantics.
- (BE-Dom) adds types. The integrity label of the returned value is lowered, taking into account the integrity labels of the two parameters.
- (B-Load) adds the integrity label of the URL and the hon flag to the request. Additionally, the request is marked as a high integrity sync action. This means that all load events have to be processed in sync between the attacked and unattacked system. The integrity label of the browser state the integrity label of the URL and the attacked mode is honest.
- (B-Include) uses the join of the browser integrity label and the URL’s integrity label, as well as the browser’s current attack state as annotations on the request. The event’s sync integrity label is the browser’s integrity label. The rule does not modify the browser’s integrity label or attacked state.
- (B-Submit) uses the integrity label and attacked mode from the DOM for the request, combined with the integrity label of the target URL. The rule does not modify the browser’s integrity label or attacked state.
- (B-RecvLoad) receives a response to a load event, labelled with an integrity label and an attacked state, and uses these labels for the DOM and the browser state. The sync integrity of the response event is the integrity label of the browser. This means that event direct responses to a load or a submit have to be processed in sync (since they leave the browser in a high integrity state). A redirect however can lower the integrity of a browser that is awaiting a response to a load or submit (see below).
- (B-RecvInclude) joins the integrity label and attacked state of the current browser state with the ones from the network response and uses them in the continuation. The sync integrity label of the event is the meet of the integrity label of the reply and the integrity label of the browser. This means that as long as one of the two is high, the response to the include has to be processed in sync.
- (B-Redirect) uses the integrity label and attacked state of the incoming event for the outgoing event and the resulting browser state. The sync integrity label of the event is the meet of the integrity label of the reply and the integrity label of the browser. This means that as long as one of the two is high, the response to the include has to be processed in sync.
- (B-Flush) sends out the event from the buffer together with its sync integrity label.
- (B-End) resets the browser’s integrity label to high integrity and resets the attacked mode to hon. The sync integrity label is high, meaning that this step always has to be processed in sync.
- (B-Seq) propagates the labels from the subcommand.
- (B-Skip) propagates the browser annotations. The sync integrity label is the integrity label of the browser state.
- (B-SetReference) evaluates the expression and stores it in the memory, with the join of computed type and the type of the reference in the typing environment \(\Gamma\).
- (B-SetDom) updates the DOM labelling by joining its original integrity label and the attacked state with the ones of the browser state. The integrity label of the value stored into the DOM is lowered using the integrity label of the browser.

2) Detailed Explanation of Changes to Server Semantics:

- (SE-Val) also contains the type.
- (SE-Fresh) samples names from the partition of the set of names indicated by the annotation. If the browser id is not the one of the honest user usr, then we always sample from \(\mathcal{N}_0\), the set of names of low confidentiality and integrity.
- (SE-BinOp) is just like (BE-BinOp).
- (SE-ReadGlobal), (SE-ReadSession) look up the reference in the corresponding part of the database.
- (S-Seq) just propagates the annotations.
- (S-IfTrue), (S-IfFalse) lower the integrity label, based on the type of the guard. In case the code for the branch does not contain any command that can lead to a response, a reset command is added after the branch, to bring the integrity label back to its original value.
TABLE XI: Extended semantics of server commands.

(S-SEQ) 
\[
(D, \phi, [c']_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D', \phi', [c'']_{\mu, \Gamma})
\]
\[
(D, \phi, [c; c'']_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D', \phi', [c; c'']_{\mu, \Gamma})
\]

(S-IFTRUE) 
\[
c'' = (\text{reply, redirect, tokencheck, originalcheck} \in \text{coms}(c)) \ ? \ c : \text{reset } l
\]
\[
eval_g(se, D) = \text{true}^\tau
\]
\[
(D, \phi, [\text{if } \text{then } c \text{ else } c'']_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [c'']_{\mu, \Gamma})
\]

(S-IFFALSE) 
\[
c'' = (\text{reply, redirect, tokencheck, originalcheck} \in \text{coms}(c)) \ ? \ c : \text{reset } l
\]
\[
eval_g(se, D) = \text{false}^\tau
\]
\[
(D, \phi, [\text{if } \text{then } c \text{ else } c'']_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [c']_{\mu, \Gamma})
\]

(S-SKIP) 
\[
(D, \phi, \text{[skip]}_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [c']_{\mu, \Gamma})
\]

(S-TCFALSE) 
\[
eval_g(se, D) = v''
\]
\[
eval_g(se', D) = v'''
\]
\[
v \neq v'
\]
\[
(D, \phi, [\text{tokencheck}(se', se)] \ ? \ c'_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{reply(update, skip, )}]_{\mu, \Gamma})
\]

(S-RECV) 
\[
\alpha = \text{req}(t, n, u, p, c, o)_{\mu, \Gamma}
\]
\[
i \in N
\]
\[
\forall k \in [1 \ldots |T|], M(r(x)) = (r(y) \in \text{dom}(ck(x)), ck(x), o, k)_{\Gamma} \quad i \in N
\]
\[
\sigma = [x_i \mapsto x_{i+1}, x_{i+1} \mapsto x_i]
\]
\[
(D, \phi, \{i \mapsto M, \phi', \{c \mapsto r\}_{(i,i+1)} \}) \xrightarrow{\alpha_{\Omega, \Gamma}}
\]
\[
(D \oplus \{i \mapsto M, \phi', \{c \mapsto r\}_{(i,i+1)} \}) \cup \{i \mapsto r\}_{(i,i+1)}
\]

(S-NEWSESSION) 
\[
E = i, \_\quad \xrightarrow{\alpha} \quad n_{\phi, \Gamma}
\]
\[
(D, \phi, [\text{start session}]_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{skip}]_{\mu, \Gamma})
\]

(S-OchkFAIL) 
\[
(D, \phi, \text{[if originchk(K)] then } c'_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, \text{[reply(error, skip, )}]_{\mu, \Gamma})
\]

(S-SETSESSION) 
\[
eval_g(se, D) = v
\]
\[
(D, \phi, [\text{tokencheck}(se, se)] \ ? \ c'_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{reply(update, skip, )}]_{\mu, \Gamma})
\]

(S-LPARALLEL) 
\[
(D, \phi, t') \xrightarrow{\alpha_{\Omega, \Gamma}} (D', \phi', t''')
\]
\[
(D, \phi, t' || t'') \xrightarrow{\alpha_{\Omega, \Gamma}} (D', \phi', t'' || t''')
\]

(S-RESTART) 
\[
(D, \phi, [\text{reset } t'']_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{skip}]_{\mu, \Gamma})
\]

(S-RPARALLEL) 
\[
(D, \phi, t' || t'') \xrightarrow{\alpha_{\Omega, \Gamma}} (D', \phi', t' || t'')
\]

(S-LOGIN) 
\[
eval_g(seuser, D) = \iota_s
\]
\[
eval_g(seuser, D) = n
\]
\[
(D, \phi, [\text{login} seuser, seuser, seuser, seuser]_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{skip}]_{\mu, \Gamma})
\]

(S-REDIRECT) 
\[
R = n, u, t, o \quad \Gamma(u) = \text{redirect } \Gamma
\]
\[
\sigma = [x_1 \mapsto x_2, x_2 \mapsto x_3] \quad \alpha = \text{redirect}(n, u, o)_{\Gamma}
\]
\[
(D, \phi, [\text{redirect } (u', z, c, k)]_{\mu, \Gamma}) \xrightarrow{\alpha_{\Omega, \Gamma}} (D, \phi, [\text{halt}]_{\mu, \Gamma})
\]
• (S-RESET) restores the integrity label to the provided value. The sync integrity label is the integrity label to which the reset is performed. This means that returning to a high integrity context from a low integrity context must be processed in sync.
• (S-Skip) just propagates the annotations
• (S-TCTrue), (S-TCFalse) just propagate the annotations.
• (S-RECV) takes the annotations from the request and uses them for the newly started thread.
• (S-RESTORESESSION) just propagates the annotations.
• (S-NEWSESSION) initializes the new memory with \( \bot \), annotated with the appropriate type from \( \Gamma_{\mathcal{R}^s} \) combined with the type of the session identifier. The integrity label is not influenced, as by an invariant the integrity of all session memory references and the user identity is upper bounded by the integrity of the session identifier
• (S-OCHKSUCCESS), (S-OCHKFFAIL) just propagate the annotations.
• (S=LPARALLEL), (R-LPARALLEL) just propagate the labelling of the events of sub threads
• (S-SETGLOBAL) stores the value with its computed type, joining the integrity label with the thread’s integrity label.
• (S-SETSESSION) stores the value with the type that results from joining the value’s original type with the type of the reference, limited by the type of the session identifier. We will show in the proof that typing then ensures that the types of values in a memory reference is always equal to the type for that reference in the environment \( \Gamma \), limited by the type of the session identifier.
• (S-LOGIN) just propagates the annotations
• (S-AUTH) just propagates the annotations
• (S-REPLY), uses the annotations of the current thread for the reply, where the integrity label is joined with the expected integrity label for the reply. In case the reply is an error message, instead of going to the regular \( \text{halt} \) state, the thread will go to a \text{bad} state. These two states are semantically equivalent (both cannot be processed further) and are just used to establish an invariant in the proofs.
• (S-REDIRECT) uses the annotations of the current thread for the reply where again the integrity label is joined with the expected integrity label for the reply.

3) Detailed Explanation of Changes to the Semantics of Web Systems with the attacker: For the proof it is required that every rule only performs a single step in a browser. We hence have to split up the rule \((A\text{-TIMEOUT})\) into two separate rules. For this reason we introduce a buffer \( T_O \) that stores the request that requires the timeout-response. As long as this buffer contains an element, the only rule that can be taken is \((A\text{-TIMEOUTRECV})\).
• (W-LPARALLEL), (W-RPARALLEL) and
• \((A\text{-NIL})\) simply propagate the annotations.
• \((A\text{-BROWSERSERVER})\) “forwards” the request with the same annotations. We use the sync label of the browser event for the event in the websystem and use the integrity label of the browser event as the sync label for the server event. This means that in some cases (for example for a load to a URL of low integrity) we will require that the browser step is performed in sync, while the server step must not be in sync, we just require that the request is processed in some form. For example, it is possible to match a server receiving a low integrity request with a case where the attacker interferes.
• \((A\text{-SERVERBROWSER})\) does the same in the other direction. Again we use the browser event’s sync integrity label for the websystem event. This allows us to synchronize two browsers receiving a low integrity a response to a load request with high sync integrity label, without synchronizing the server step. For example we can match a server responding to the request with the attacker responding to the request.
• \((A\text{-TIMEOUTSEND})\) \((A\text{-TIMEOUTRECV})\) are two individual rules that together equivalent to the rule \((A\text{-TIMEOUT})\). In rule \((A=\text{TIMEOUTSEND})\) all relevant information is stored in the buffer \( T_O \) so that rule \((A\text{-TIMEOUTRECV})\) can send the corresponding response. Note that the integrity label and the sync integrity label may be different.
• \((A\text{-BAOATK})\)“forwards” the request with the same annotations.
• \((A\text{-ATKSER})\) sends an event labelled with low integrity and attacker mode att and annotated with low integrity.
• \((A\text{-SERATK})\) “forwards” the request with the same annotations.
• \((A\text{-ATKBBRO})\) creates a response with low integrity and attacked mode att. The event can have any sync integrity label – since the browser may expect a different label in different situations.

We show that the original semantics and the extended semantics are equivalent for well typed fresh web systems. Concretely we show that they can produce the same traces. We use here the notation for well-typed websystems \( \Gamma \models_{\mathcal{F},\mathcal{U},\mathcal{S}} A \), that is formally introduced in definition \( \text{[13]} \).

Lemma 1 (Semantic Equivalence). Let \( A \) be a fresh web system with \( \Gamma \models_{\mathcal{F},\mathcal{U},\mathcal{S}} A \).

1) if for some \( \vec{\alpha}, A' \) we have \( A \xrightarrow{\vec{\alpha}} A' \) then there exists \( A'' \) such that \( A \xrightarrow{\vec{\alpha}} A'' \),
2) if for some \( \vec{\alpha}, A' \) we have \( A \xrightarrow{\vec{\alpha}} A' \) then there exists \( A'' \) such that \( A \xrightarrow{\vec{\alpha}} A'' \),

Proof. The claim follows directly by induction over the derivation of \( \vec{\alpha} \), using the following observations:
We also present additional or modified typing rules, that allow us to type situations occurring only at runtime. As a convention we use $\vdash$ for the extended typing judgements, while we use $\vdash$ for the original typing judgements. New rules with the same name as an original rule replace that rule, all other original rules also become new rules without modification. Rules with new names are additional rules.

4) Detailed explanation of changes to Typing Rules:

- **(T-EFRESH)** assigns the type $\ell_a$ to a fresh() expression if it is typed in the attacker’s run. \\
- **(T-RUNNING)** allows us to type running server threads. The typing branch is determined based on the browser identity and the attacked mode of the thread. The typing environment for global variables is determined by the browser identity. If it is the honest users’ browser, then the original typing environment is used (since the cookies come from the honest browser). Otherwise, we use an environment where every type is $\ell_a$. We then type the code of the thread, inferring the session label $jlabel(j)$ from the session identifier $j$ and using the integrity label as $pc$. \\

### TABLE XII: Extended semantics of web systems with the attacker.

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Extended Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(W-LPARALLEL)</strong></td>
<td>$W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$</td>
</tr>
<tr>
<td></td>
<td>$W \parallel W'' \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W' \parallel W''$</td>
</tr>
<tr>
<td><strong>(W-RPARALLEL)</strong></td>
<td>$W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$</td>
</tr>
<tr>
<td></td>
<td>$W'' \parallel W' \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'' \parallel W'$</td>
</tr>
<tr>
<td><strong>(A-NIL)</strong></td>
<td>$T_O = { }$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\alpha \in { _ }$</td>
</tr>
<tr>
<td></td>
<td>$(\ell_a, K) \vdash W \frac{\alpha \rightarrow \Gamma}{\vdash} (\ell_a, K) \vdash W'$</td>
</tr>
</tbody>
</table>

- **(A-BROWSER_SERVER)**

  $T_O = \{ \}$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **(A-SERVER_BROWSER)**

  $T_O = \{ \}$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **(A-TIMEOUTSEND)**

  $W' \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W' \Rightarrow$ $W''$

- **(A-TIMEOUTRECV)**

  $T_O = \{ (\ell_b, n, u, l, \mu) \}$ $T_O' = \{ \}$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$$\Rightarrow$ $W''$

- **(A-BROATK)**

  $T_O = \{ \} \alpha = \text{res}(\ell_b, n, u, p, c, o)_{\mu, 1} \Rightarrow W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **(A-ATKSER)**

  $T_O = \{ \}$ $n \leftarrow N$ $\Rightarrow$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **(A-SERATK)**

  $T_O = \{ \}$ $n \in K$ $\Rightarrow$ $W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **(A-ATK BRO)**

  $T_O = \{ \}$ $\alpha = \text{res}(\ell_b, n, u, r', \vec{v}, c, page, s)_{\mu, 1} \Rightarrow W \frac{\alpha \rightarrow \Gamma}{\vdash}$ $W'$ $\Rightarrow$ $W''$

- **The integrity label and the attacker state are simply annotations and do not prevent or allow additional steps in the semantics.**
- **The same is true for the type annotations on values, however we must prevent certain joins on credential types, as they are not defined. Typing ensures that these cases don’t occur.**
- **The command reset $l$ is just modifying the integrity label of the thread, but is otherwise a no-op (S-RESET), so adding it in (T-IFF=true) and (T-IFFFALSE) does not impact the behaviour of the program.**
- **The split of (A-TIMEOUT) into two separate rules does not impact the semantics as no other rule can be used as long as there is a pending timeout response in the buffer $T_O$.**

We also present additional or modified typing rules, that allow us to type situations occurring only at runtime. As a convention we use $\vdash$ for the extended typing judgements, while we use $\vdash$ for the original typing judgements. New rules with the same name as an original rule replace that rule, all other original rules also become new rules without modification. Rules with new names are additional rules.

4) Detailed explanation of changes to Typing Rules:

- **(T-EFRESH)** assigns the type $\ell_a$ to a fresh() expression if it is typed in the attacker’s run.

- **(T-RUNNING)** allows us to type running server threads. The typing branch is determined based on the browser identity and the attacked mode of the thread. The typing environment for global variables is determined by the browser identity. If it is the honest users’ browser, then the original typing environment is used (since the cookies come from the honest browser). Otherwise, we use an environment where every type is $\ell_a$. We then type the code of the thread, inferring the session label $jlabel(j)$ from the session identifier $j$ and using the integrity label as $pc$. 


The addition of other rules does not impact the claim. The changes in the rule $(T-EF)$ also have

$$\delta = \begin{cases} \text{fresh} \end{cases} \Rightarrow \Gamma \vdash \ell : \tau$$

Proof. The proof follows by induction on the typing derivation using the following observations:

- $(T-EF)$ now gives values their annotated type.
- $(T-AUTHATT)$ does not perform any checks for authenticated events when typing the attackers branch.
- $(T-HALT)$ trivially checks the $halt$ and $bad$ commands (which only occur at runtime).
- $(T-REPLY)$ now only requires the script to be well typed if we are not typing the attackers branch (i.e., only if the script is sent to the honest user’s browser) and additionally passes the URL to the typing judgements for scripts.
- $(T-REPLYERR)$ trivially checks the response with an error message.
- $(T-RESET)$ raises the $pc$ for the continuation to the label provided in the reset statement.
- $(T-BERFAIL)$ now gives values their annotated type.
- $(T-BERFAIL)$ allows us to give type any type $\tau$ to a browser reference if it may not be read by the script. This rule (and the next one) is needed to ensure that scripts provided by the attacker can be typed (although they will not execute correctly).
- $(T-BASSIGNFAIL)$ allows us to type any assignment to a browser reference, if the script is not allowed to write to it.

We now show that typing with the original typing rules implies typing with the extended rules.

Lemma 2 (Typing Equivalence). For any fresh server $S = (\{\}, \{t\}$, whenever we have $\Gamma, \ell, s, pc \vdash \ell : \ell, pc$ then we also have $\Gamma, \ell, s, pc \vdash \ell : \ell, pc$.

Proof. The proof follows by induction on the typing derivation using the following observations:

- Every typing rule in the original system is also a typing rule in the extended system, with the exception of the modified rules $(T-EF)$, $(T-EVAL)$, $(T-REPLY)$, $(T-BEVAL)$.
- The changes in rules $(T-EF)$ and $(T-BEVAL)$ return the type annotations, which are according to the definition of $\downarrow$ for values $v \notin N$. Thus the result is the same as in the original typing rule.
- The addition of other rules does not impact the claim.

J. Subject Reduction

In this section we prove subject reduction for the web system. This is needed to ensure that the system is always in a well-typed state, which in turn is required to prove that our high integrity relation is preserved.

We look at typing of different components of the web system individually. Concretely we will define typing for requests and responses, browsers, servers and webservers as a whole.
We start by defining well-typed requests and responses. Then we define well-typed browsers and show that typing is preserved when the browser takes a step, if the browser only receives well-typed responses, and show that the browser only sends out well-typed requests. We then define well-typed servers and show that typing is preserved whenever the server takes a step, if all requests received by the server are well typed, and that all responses produced by the server are well-typed. We furthermore show that all requests and responses produced by the attacker are well-typed. Finally, we define well-typed web-systems and show that typing is preserved whenever the web-system takes a step.

**Definition 9 (Request Typing).** For a request $\alpha = \text{req}(t_b, n, u, p, c_k, o)^{\ell, \tau, l}$ (resp. $\alpha = \text{req}(t_b, n, u, p, c_k, o)^{\ell, \tau, l}$) with $\Gamma_{ul}(u) = \ell_u, \tau, l$, we have $\Gamma \vdash_{\ell_u, \text{usr}} \alpha$ if

1. if $\mu = \text{hon}$ and $t_b = \text{usr}$ then
   - for all $k \in \text{dom}(p)$ we have if $p(k) = v_k^\ell$ then $\tau_k \subseteq \ell_a \tau_k$
   - $l \subseteq_I (\ell_u)$
2. if $\mu = \text{att}$ then
   - for all $k \in \text{dom}(p)$ we have if $p(k) = v_k^\ell$ then $\tau_k' \subseteq \ell_a$
   - $l \subseteq_I (\ell_u)$
3. if $t_b = \text{usr}$
   - for all $c \in \text{dom}(c_k)$ we have
     - if $c_k(c) = v_c^\ell$ then $\tau_c \subseteq \ell_a \text{ref}_c(\Gamma_{u}o(c))$
     - $C(\lambda(v)) \subseteq \ell_a C(\lambda(a))$
4. if $t_b \neq \text{usr}$ then for all $c \in \text{dom}(c_k)$ we have if $c_k(c) = v_c^\ell$ then $\tau_c \subseteq \ell_a$
5. If $t_b = \text{usr}, u \in \mathcal{P}$ and $o \neq \bot$ and $I(\ell_a) \not\subseteq I(o)$ then $\mu = \text{hon}$.

Intuitively, according to definition 9 a request is well-typed if

1. for all honest requests, all parameter types are respected and the integrity label is higher than the integrity label of the URL.
2. for all attacked requests, all parameters are of the attacker’s type and the integrity is low.
3. For all (attacked and honest) requests from the users browser, all cookies respect their type from the environment and their confidentiality is as most as high as the one of the URL.
4. For all requests by the attacker, all cookies have the type of the attacker.
5. Any request with a high integrity origin to a protected URL must be honest.

We now in a similar fashion define well-typed responses.

**Definition 10 (Response Typing).** For a response $\alpha = \text{res}(t_b, n, u, u', \vec{v}, c_k, page, s)^{\ell, \tau, l}$ (resp. $\alpha = \text{res}(t_b, n, u, u', \vec{v}, c_k, page, s)^{\ell, \tau, l}$) with $\Gamma_{ul}(u) = \ell_u, \vec{\tau}, l_r$, we have $\Gamma \vdash_{\ell_u, \text{usr}} \alpha$ if

1. For all $v^\vec{\tau} \in \text{values}(c_k, \text{page}, s, \vec{v})$ we have $C(\tau) \subseteq C(\ell_u)$
2. If $t_b \neq \text{usr}$, then for all $v^\vec{\tau} \in \text{values}(c_k, \text{page}, s, \vec{v})$ we have $\tau \subseteq \ell_a$
3. If $t_b = \text{usr}$ and $\mu = \text{att}$ then for all $u''$ with $I(\ell_u) \subseteq I(\lambda(u''))$ we have $\Gamma, \text{att}, T_I \vdash_{\ell_u, \mathcal{P}, u''} s$ and $I(\ell_a) \subseteq_I l$
4. If $t_b = \text{usr}$ then for all $v \in \text{dom}(\text{ck})$ with $\text{ck}(r) = v^\vec{\tau}$ we have
   - $\Gamma(\lambda(u)) \subseteq_I \lambda(r)$ then $\tau \subseteq \ell_a \text{ref}_r(\Gamma_{u}o(r))$ and $l \subseteq_I \lambda(\text{ref}_r(\Gamma_{u}o(r)))$
   - $\Gamma(\lambda(u)) \not\subseteq_I \lambda(r)$ then $\tau \subseteq \ell_a o$ and $\tau \subseteq \ell_a$
5. If $t_b = \text{usr}$ and $I(\ell_a) \not\subseteq_I l$ then $\text{page} = \text{error}$ or for all $v \in \text{dom}(\text{page})$ with $\text{page}(v) = \text{form}(u'', \vec{v})$ we have
   - $\Gamma_{l}(u'') = \Gamma_{\vec{v}}(v)$
   - with $\Gamma_{ul}(u'') = \ell_u, \vec{\tau}, l_r$
   - for all $i \in [1 \ldots |\vec{v}|]$ we have $\tau_i \subseteq \ell_a \tau_i'$
   - $l \subseteq_I \ell_u$
6. If $t_b = \text{usr}$ and $I(\ell_a) \not\subseteq_I l$ then $\text{page} = \text{error}$ or we have one of the following
   - for all $v \in \text{dom}(\text{page})$ with $\text{page}(v) = \text{form}(u'', \vec{v})$, for all $i \in [1 \ldots |\vec{v}|]$ we have $\tau_i \subseteq \ell_a \ell_a$
   - or $I(\ell_a) \not\subseteq_I \ell_u$
7. $t_b = \text{usr}$ and $\mu = \text{hon}$ then $\Gamma, \text{hon}, l_r \vdash_{\ell_u, \mathcal{P}, u} s$ and $l = l_r$
8. If $u' \neq \bot$ and $t_b = \text{usr}$ then with $\alpha' = \text{req}(t_b, n', u', p, \{\}, \bot)^{\ell, \vec{v}, l}$, for any $n'$ and $\forall k \in [1 \ldots |\vec{v}|] : p(k) = v_k$ we have $\Gamma \vdash_{\ell_u, \text{usr}} \alpha'$.

Intuitively, according to definition 10 a response is well-typed if

1. The confidentiality label of all values contained in the response is at most as high as the confidentiality label of the URL from which the response is sent, or the confidentiality is low.
2) If the response it not sent to the honest user, then all values must be of low confidentiality.
3) If the response is sent to the honest user and influenced by the attacker, then the integrity label is low and the contained script is well-typed, using the type branch att.
4) For all responses to the honest users, if a cookie may be set by the response, then it respects the typing environment (also taking into account the integrity label of the response). If the cookie may not be set by the response, then it respects the typing environment or is low.
5) For all honest responses we have that the page is either the error page, or that it is well-typed, i.e., the type of the form will result from processing the response at the browser is well typed (using an empty set of cookies and an empty origin as placeholders).

**Definition 11 (Browser Typing).** Let $B = (N, M, P, T, Q, a)^{i,u,j}_{\mu}$ be a browser. We write $\Gamma \models_{\ell, u, \mu} B$, if $\ell_b = u$ and

1) $\mu = \text{att} \Rightarrow I(\ell_a) \subseteq I l$
2) $\forall r \in \text{dom}(M), M(r) = \nu^r \land \tau = \text{ref}(\Gamma_{\ell_a}(r))$
3) For all tab $\in \text{dom}(P) = P(tab) = (u, page, l', \mu')$ and page $\neq$ error we have for all $v \in \text{dom}(page)$ with page$(v) =$ form$(u^r, \nu^r, \tau^r)$

- $C(\nu^r) \subseteq C(\ell_a)$
- if $I(\ell_a) \not\subseteq l \land I(\ell_a)$ then
  - $\mu = \text{hon}$
  - $\Gamma_{u'}(u') = \Gamma_X(v)$
  - $l' \subseteq I (\ell_a)$
  - $\forall i \in [1 . . . |\nu^r|], \tau_i \subseteq \ell_a, \tau'_i$
- if $I(\ell_a) \subseteq I l$ then one of the following holds
  - $\forall i \in [1 . . . |\nu^r|], C(\tau_i) \subseteq C(\ell_a)$
  - $I(\ell_a) \not\subseteq I (\lambda(u))$

4) If $T = \{\text{tab} \rightarrow s\}$ and $P(tab) = (u, page, l', \mu')$ with $\Gamma_{u'}(u') = \ell_u, \tau_u, l_r$ then

- $l = l'$
- if $\mu = \text{hon}$ then $l' \cup I l \subseteq I l_r$ and $\Gamma, \text{hon}, l_r \models_{\ell_a, p, s}$
- if $\mu = \text{att}$ then $\Gamma, \text{att}, T_l \models_{\ell_a, p, s}$

5) If $Q = \{\alpha \not\in \ell'\}$, then we have $\Gamma \models_{\ell_a, u, s, \alpha}$.

6) For $\ell$ we have

- for the navigation flow $\ell'$ in $\ell$, we have that $I(\ell_a) \subseteq I (\lambda(a'_j))$ implies $I(\ell_a) \subseteq I (\lambda(a'_k))$ for all $j < k \leq |\ell_a'|$.
- if $N = \{n \rightarrow (\text{tab}, u, o)\}$ and $T = \{\}$ then we have that for all $a \in \text{nf}(\ell, \text{tab})$ that $I(\ell_a) \subseteq I l$ implies $I(\ell_a) \subseteq I (\lambda(a))$. Furthermore we have $I(\ell_a) \subseteq I (\lambda(a'_j))$ implies $I(\ell_a) \subseteq I (\lambda(a'_k))$ for all $j < k \leq |\ell_a'|$.
- for all tab $\in \text{dom}(P)$ with $P(tab) = (u, page, l', \mu')$ and $N \not= \{n \rightarrow (\text{tab}, u, o, n)\}$ for all $n, u, o, n$, we have that for all $a \in \text{nf}(\ell, \text{tab})$ that $I(\ell_a) \subseteq I (\lambda(u))$ or $\mu' = \text{att}$ implies $I(\ell_a) \subseteq I (\lambda(a))$. Furthermore we have $I(\ell_a) \subseteq I (\lambda(a'_j))$ implies $I(\ell_a) \subseteq I (\lambda(a'_k))$ for all $j < k \leq |\ell_a'|$.

- for all actions $a'$ in $\ell$ we have:
  - if $a' = \text{load}(\text{tab}, u, p)$ and $\Gamma_{u'}(u') = \ell_u, \tau_u, l_r$ then for all $k \in \text{dom}(p)$ we have that if $p(k) = \nu^r$ then $\tau' \subseteq \ell_u, \tau_k$;
  - if $a' = \text{submit}(\text{tab}, u, v', p)$ and $\Gamma_{u'}(v') = \ell_u, \tau_u, l_r$ then for all $k \in \text{dom}(p)$ we have that if $p(k) = \nu^r$ then $\tau' \subseteq \ell_u, \tau_k$;

- if $I(\ell_a) \subseteq I \lambda(u)$ then additionally $C(\tau') \subseteq C(\ell_a)$

7) If $N = \{n \rightarrow (\text{tab}, u, o)\}$ and $T = \{\text{tab} \rightarrow s\}$ and $\mu = \text{hon}$ then

- if $P(tab) = (u', page, l', \mu')$ and $\Gamma_{u'}(u') = \ell_u, \tau_u, l_r$ and $\Gamma_{u'}(u') = \ell'_u, \tau'_u, l'_r$ then $l_r = l'_r$.
- $I(\ell_a) \not\subseteq I (\lambda(u))$
Intuitively, according to definition 1, a browser is well-typed, if all its components are well-typed. Concretely, we require that:
1) Whenever the state of the browser is directly influenced by the attacker, then the integrity of the browser is low.
2) All values stored in a memory reference have a type annotation that is equal to the type of the reference the typing environment.
3) For any non-empty DOM in a tab,
   • If the DOM is of high integrity
     – The DOM is honest
     – The type of the form name matches the type of the URL
     – The integrity label of the DOM is higher than the integrity label of the URL
     – All parameters have the expected type.
   • If the DOM is low integrity
     – and either
       * All parameters have the attacker’s type.
       * or the integrity of the DOM’s origin is high
4) If a script is running in a tab
   • the script integrity is equal to the integrity of the DOM in that tab.
   • if the browser is not attacked then the browser’s integrity is equal the integrity of the expected response type for the URL of the DOM in the same tab and the script code is well-typed in the honest typing branch using the expected response type as the pc.
   • if the browser is attacked, then the script is well-typed using $T_I$ label as pc.
5) All requests in the buffer are well-typed.
6) For all user actions we have that
   • The user will not submit forms on high integrity pages after “tainting” the connection, by visiting a low integrity page. Concretely the conditions are the following:
     – The first condition is exactly the assumption we make on well-formed user actions.
     – The second condition is the same, but taking into account open network connections for load or submits.
     – The third condition is similarly taking into account pages already loaded in browser tabs for the navigation flow.
   • The user’s inputs respect the parameter types and will only input low confidentiality values in forms present in low integrity pages.
7) Whenever the browser is in an honest state, has a script running in the context of URL $u$ and is waiting for the response of a script inclusion from URL $u'$, then
   • the two URLs have the same expected response type.
   • the URL $u'$ is of high integrity.

We now prove that whenever a browser expression containing variables is well typed in a typing environment, then it is also well-typed if we substitute the variables with concrete values of the expected type.

**Lemma 3** (Browser Expression Substitution). Whenever we have $\Gamma, b \vdash \sigma \in_{\ell_a} \Gamma$ and $\forall x \in dom(\Gamma_X), \sigma(x) = v^x_{\ell} \text{ with } \tau_x \subseteq \ell_a \Gamma_X(x)$ then for all $\Gamma_X$, we have $(\Gamma_u, \Gamma_X, \Gamma_R^\sigma, \Gamma_R^\ell, \Gamma_V), b \vdash \sigma : \tau$.

**Proof.** We perform an induction on the typing derivation of $\Gamma, b \vdash \sigma : \tau$:
1. (T-BVAR). Then $be = x$ and $be\sigma = v^x_{\ell}$ with $\tau_x \subseteq \ell_a \Gamma_X(x)$. The claim follows directly from rule (T-BVAL) and (T-BSUB).
2. (T-BREF). Then $be = r = be\sigma$ and the claim is trivial.
3. (T-BEVAL). Then $be = v^{rv} = be\sigma$ and the claim is trivial.
4. (T-BEUNDEF). Then $be = \bot = be\sigma$ and the claim is trivial.
5. (T-BNAME). Then $be = n^{v_n} = be\sigma$ and the claim is trivial.
6. (T-BEDOM). Then $be = dom(be_1, be_2)$ and $be\sigma = dom(be_1\sigma, be_2\sigma)$ and the claim follows immediately using (T-BEDOM).
7. (T-BBINOP). Then $be = be_1 \odot be_2$ with $\Gamma, b \vdash \sigma = \tau_1$ and $\Gamma, b \vdash \sigma = \tau_2$ and $\tau = label(\tau_1) \sqcup label(\tau_2)$. We also have $be\sigma = be_1\sigma \odot be_2\sigma$. By induction we know that $\Gamma, b \vdash be_1\sigma : \tau_1'$ and $\Gamma, b \vdash be_2\sigma : \tau_2'$ with $\tau_1' \subseteq \ell_a \tau_1$ and $\tau_2' \subseteq \ell_a \tau_2$. We then know that $label(\tau_1') \sqcup label(\tau_2') \subseteq \ell_a label(\tau_1) \sqcup label(\tau_2) = \tau$, and the claim follows by (T-BINOP) and (T-BESUB).
8. (T-BESUB) follows by induction and by the transitivity of $\subseteq \ell_a$. 


Next, we prove the same claim on the level of scripts.

**Lemma 4 (Browser Substitution).** Whenever we have $\Gamma, p \vdash_b^\ell \tau, p \; s$ and we have a substitution $\sigma$ with $\text{dom}(\sigma) = \text{dom}(\Gamma_\chi)$ and $\forall x \in \text{dom}(\Gamma_\chi), \sigma(x) = v^\sigma_x$ with $\tau_x \subseteq \ell_x, \Gamma_\chi(x)$ then for all $\Gamma_\chi'$, we have $(\Gamma_\eta, \Gamma_\chi', \Gamma_\eta \triangleright \Gamma_\chi, \Gamma_\eta) \vdash_b^\ell \tau, p \; s \sigma$.

*Proof.* We do the proof by induction on the typing derivation.

- (T-BSEQ): Then $s = s_1, s_2$. The claim follows by applying the induction hypothesis to $s_1$ and $s_2$ and applying rule (T-BSEQ).
- (T-BSKIP): The claim follows trivially.
- (T-ASSIGN): Then we have $s = r := b e$, with
  - $\Gamma \vdash_b^\ell r : \text{ref}(\tau)$
  - $\Gamma, b \vdash_b^\ell \tau$.
- (T-BSETDOM): Then $s = \text{setdom}(v, u, b e)$. The claim follows by applying of lemma 3 and (T-BESUB) for every $b e_i$ in $b e$.
- (T-BINCLUDE): Then $s = \text{include}(u, b e)$. The claim follows by applying of lemma 3 and (T-BESUB) for every $b e_i$ in $b e$.

Now, we show that typing is preserved under the evaluation of expressions.

**Lemma 5 (Browser Expression Typing).** Let $B = (N, M, P, T, Q, \vec{a})^{\ell_1, \ell_2, \mu}$ be a browser with $\Gamma \vdash_{\ell_1, \text{usr}} B$. Let $T = \{\text{tab} \mapsto s\}$ and $\{\text{tab} \mapsto (u, f, \ell', \mu')\} \in P, \ell = \lambda(u)$. Then for any browser expression $b e$, if $\Gamma, \mu \vdash_b^\ell \tau$ then $\Gamma, \mu \vdash_b^\ell \text{eval}(b e, M, f) : \tau$

*Proof.* Let $\text{eval}(b e, M, f) = v^{\tau'}$. We show $\tau' \subseteq_{\ell_a} \tau$ and the claim follows using rule (T-BESUB). We perform the proof by induction over the expression $b e$:

- $b e = x$: In this case, $\text{eval}(x, M, f)$ is undefined, so we do not have to show anything.
- $b e = v^{\tau'}$: We have $\text{eval}(v^{\tau'}, M, f) = v^{\tau'}$ and the claim is trivial.
- $b e = b e_1 \circ b e_2$: By induction we know
  - $\Gamma, \mu \vdash_b^\ell b e_1 : \tau_1$ and $\text{eval}(b e_1, M, f) = v_1^{\tau_1}$ and $\tau_1' \subseteq_{\ell_a} \tau_1$
  - $\Gamma, \mu \vdash_b^\ell b e_2 : \tau_2$ and $\text{eval}(b e_2, M, f) = v_2^{\tau_2}$ and $\tau_2' \subseteq_{\ell_a} \tau_2$

Let now $v^{\tau'} = v_1^{\tau_1} \circ v_2^{\tau_2}$. Then we know that $\tau' = \text{label}(\tau_1') \sqcup \text{label}(\tau_2')$ by rule (BE-BINOP). By rule (T-BEBINOP) we have $\tau = \text{label}(\tau_1) \sqcup \text{label}(\tau_2)$, and the claim follows.

- $b e = r$: then the claim immediately follows from rule (T-BEREF) and property 2 of definition 11
- $b e = \text{dom}(b e_1, b e_2)$: We know by property 3 of definition 11 $l = l'$. We distinguish two cases:
  - If $I(\ell_a) \not\subseteq I l'$ then we know that $\mu = \text{hon}$ and hence this case is impossible, since we do not have a typing rule for the expression in the honest type branch.
  - If $I(\ell_a) \subseteq I l'$, then we distinguish two cases:
    - * If $I(\ell_a) \not\subseteq I l(\lambda(u))$ then we know that the script can also be typed with $b = \text{hon}$, and hence this case is impossible.
    - * If $I(\ell_a) \subseteq I l(\lambda(U))$ then by rule (BE-DOM) the value is either a URL parameter or the URL itself. We then know that for all parameters $v^{\tau''}$ of any URL in the DOM we have $C(\tau') \subseteq_{C} C(\ell_a)$. For any URL $u^{\tau''}$ we have $C(\tau_u) \subseteq_{C} C(\ell_a)$ and the claim holds.

We now show subject reduction for the browser for internal steps i.e., whenever a well-typed browser takes a step, it results in another well-typed browser. We treat browsers sending requests and receiving responses in separate lemmas.

**Lemma 6 (Browser Subject Reduction).** Let $B, B'$ be browsers with $\Gamma \vdash_{\ell_a, \text{usr}} B$ such that $B \xrightarrow{\ell_0} B'$. Then we have $\Gamma \vdash_{\ell_a, \text{usr}} B'$.

*Proof.* Let $B = (N, M, P, T, Q, \vec{a})^{\ell_1, \ell_2, \mu}$ and $B' = (N', M', P', T', Q', \vec{a}')^{\ell_1', \ell_2', \mu'}$ be browsers as in the lemma. We know that $t_b = \text{usr}$ and do a proof by induction on the step taken. We show that all properties of definition 11 hold for $B'$.

- (B-LOAD): Property 1 is trivial, since $\mu' = \text{hon}$.
– Property 2 is trivial, since $M = M'$
– Property 3 is trivial, since $P = P'$
– Property 4 is trivial, since $T = \{\}$.  
– For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \mathsf{rec}(t_b, n, u, p, ck, \bot, \lambda(\alpha))^{\mathcal{I}(\lambda(u))}$ and hence have to show that $\Gamma \models_{\ell_a, \usr} \alpha$. We show that all the properties of definition 9 are fulfilled.

* Property 1 follows immediately from property 6 of definition 11 for $B$
* Property 2 is trivial since we have $\mu = \text{hon}$
* Property 3 follows immediately from property 2 of definition 11 for $B$ and the definition of $\mathsf{get}(\cdot, \cdot)$.
* Property 4 is trivial since $t_b = \usr$
* Property 5 is trivial since $o = \bot$.

– Property 6 for $B'$ follows directly from property 6 of definition 11 for $B$. The navigation flow started by the load action is the same as $\text{nf}((\alpha, \mathsf{tab}))$.
– Property 7 is trivial since $T' = \{\}$

• (B-INCLUDE)

– Property 1 is trivial, since $\mu' = \mu$ and $l = l'$
– Property 2 is trivial, since $M = M'$
– Property 3 is trivial, since $P = P'$
– Property 4 is trivial using rule (T-BSKIP), since $T = \{\text{tab} \mapsto \text{skip}\}$.
– For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \mathsf{rec}(t_b, n, u, p, ck, \text{orig}(u'))^{\mathcal{I}(\lambda(u'))}$ and hence have to show that $\Gamma \models_{\ell_a, \usr} \alpha$. We show that all the properties of definition 9 are fulfilled.

* For property 1 We distinguish two cases:
  1) if $\mu = \text{hon}$ then it follows from property 4 of definition 11 for $B$ using rule (T-BINCLUDE) and lemma 5
  2) if $\mu = \text{att}$ then the claim is trivial
* For property 2 We distinguish two cases:
  1) if $\mu = \text{hon}$ then the claim is trivial
  2) if $\mu = \text{att}$ then it follows from property 4 of definition 11 for $B$ using rule (T-BINCLUDE) and lemma 5
* Property 3 follows immediately from property 2 of definition 11 for $B$
* Property 4 is trivial since $t_b = \usr$
* For property 5 we perform a case distinction:

  - If $v \not\in P$, $I(\ell_a) \nsubseteq I \text{orig}(u')$ or $\mu' = \text{hon}$ then the claim is trivial.
  - If $u \in P$, $I(\ell_a) \nsubseteq I \text{orig}(u')$ and $\mu' = \text{att}$ then assume that the include statement is contained in the script $s_{u'}$ served by $u'$. Since $u'$ is of high integrity, we know that the script code can be typed with $b = \text{hon}$. This in particular implies that every include statement in the script also has been typed with $b = \text{hon}$. Hence we know by rule (T-BINCLUDE) that $u \not\in P$ and we have a contradiction. If the include statement is not contained in the script $s_{u'}$ served by $u'$, then it must be contained in the script $s_{u''}$ served from some URL $u''$ that is included by the script $s_{u'}$. Using the same argumentation, we know by rule (T-BINCLUDE) that $I(\ell_a) \nsubseteq I \text{orig}(u'')$ and again using the same argumentation we get the contradiction $u \not\in P$

– Property 6 of definition 11 for $B'$ follows directly from property 6 for $B$.
– Property 7 follows from property 4 of definition 11 for $B$ using rule (T-BINCLUDE)

• (B-SUBMIT) Then we have

  – $a = \mathsf{submit}(\text{tab}, u, v, p')$
  – $\{\text{tab} \mapsto (u, f, l', \mu')\} \in P$
  – $\{v \mapsto \text{form}(u', v')\} \in f$
  – $\forall k \in \{1, \ldots, |v'|\}, p(k) = k \in \text{dom}(p') \Rightarrow p'(k) : v'_{k=}$.

Property 1 follows from property 4 of definition 11 for $B$.
– Property 2 is trivial, since $M = M'$
– Property 3 is trivial, since $P = P'$
– Property 4 is trivial, since $T = \{\}$
– For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \mathsf{rec}(t_b, n, u', p, ck, \text{orig}(u))^{\mathcal{I}(\lambda(u'))}$ and hence have to show that $\Gamma \models_{\ell_a, \usr} \alpha$. We show that all the properties of definition 9 are fulfilled.

* For property 1 we distinguish two cases:
  1) if $\mu' = \text{hon}$ then it follows from property 5 and 6 of definition 11 for $B$
  2) if $\mu' = \text{att}$ then we distinguish two cases:
· if \( I(\ell_a) \subseteq_I I(\lambda(u)) \) then the claim is trivial.
· otherwise, we know from property 6 that \( I(\ell_a) \subseteq_I \lambda(a) \). By the definition of \( \lambda \) we get \( \lambda(a) = \lambda(u) \) which is a contradiction to our assumption. Hence this case cannot happen.

* For property 2, we distinguish two cases:
  1) if \( \mu' = \text{hon} \) then the claim is trivial.
  2) if \( \mu' = \text{att} \) then it follows from property 3 and 6 of definition 11 for \( B' \).

* Property 5 follows immediately from property 2 of definition 11 for \( B \) and lemma 8.

* Property 4 is trivial since \( l_{\mu} = \mu \).

* For property 3 we perform a case distinction:
  · If \( u' \notin \mathcal{P}, I(\ell_a) \subseteq_I \text{orig}(u) \) or \( \mu' = \text{hon} \) then the claim is trivial.
  · If \( u' \in \mathcal{P}, I(\ell_a) \subseteq_I \text{orig}(u) \) and \( \mu' = \text{att} \) then we know \( I(\ell_a) \subseteq_I \lambda(a) \). We then get by property 6 of definition 11 for \( B \) that \( I(\ell_a) \subseteq_I \text{orig}(u) \) or \( \mu = \text{hon} \) and immediately have a contradiction.

* Property 6 of definition 11 for \( B' \) follows from property 4 for \( B \), since request from low integrity pages, are also of low integrity and since high integrity pages do not include low integrity pages (by (T-FORM)).

* Property 7 is trivial since \( T = \emptyset \).

- (B-SEQ) Then \( T = \{ \text{tab} \mapsto s \} \) with \( s = s_1; s_2 \) and from (B-BSEQ) we know \( \Gamma, \mu, l_r \vdash_{\ell_a, \mathcal{P}, \mu, u} s_1; s_2 \). We apply the induction hypothesis for the browser stepping from script \( s_1 \) to \( s_1' \). This immediately gives us all properties from definition 11 except the typing of the script \( \Gamma, \mu, l_r \vdash_{\ell_a, \mathcal{P}, \mu, s_1'; s_2} \), but this claim follows immediately by applying rule (T-BSEQ).

- (B-Skip) Then \( T = \{ \text{tab} \mapsto s \} \) with \( s = \text{skip}; s' \). By rule (T-BSEQ) we have \( \Gamma, \mu, l_r \vdash_{\ell_a, \mathcal{P}, \mu, s'} \). Since nothing besides the script changes, the claim follows immediately.

- (B-END)
  · Property 4 is trivial since \( \mu' = \text{hon} \).
  · Property 5 is trivial since \( M = M' \).
  · Property 6 is trivial since \( P = P' \).
  · Property 7 is trivial since \( T = \emptyset \).
  · Property 8 is trivial since \( Q = \emptyset \).
  · Property 9 is trivial since \( \bar{a} = \bar{a}', P = P' \) and \( N = N' \).
  · Property 10 is trivial since \( M = \emptyset \).

is trivial, since the only change from \( B \) to \( B' \) is that \( T' = \emptyset \), in which case we don’t have to show anything for the script.

- (B-SETREFERENCE) Then \( T = \{ \text{tab} \mapsto s \} \) with \( s = r :: \text{be} \). We have \( P = P' \) and claim 3 of definition 11 is trivial and since \( T' = \{ \text{tab} \mapsto \text{skip} \} \) claim 4 follows immediately from rule (T-BSKIP).

By rule (B-SETREFERENCE) we have

- \( P = P_0 \cup \{ \text{tab} \mapsto (u, f, l', \mu') \} \).
- \( \ell = \lambda(u) \).
- \( \text{eval}(\text{be}, M, f, l') = v' \).
- \( M' = M \langle r \mapsto v'^r \rangle \ni \tau_r = \tau \sqcup \text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \sqcup \text{ref}_l \).

All properties of definition 11 except for property 2 are trivial. For property 2 it is sufficient to show that \( \tau_r \subseteq_{\ell_a} \text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \).

By rule (T-BASSIGN) and rule (T-BREF) we get that

- \( \Gamma, b \vdash_{\ell_a} \text{be} : \text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \).
- \( l \subseteq_I I(\text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \).

By lemma 5 we get \( \tau \subseteq_{\ell_a} \text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \). We hence get \( \tau_r = \text{ref}_r(\Gamma_{\mathcal{R}^{\alpha}(r)}) \) and the claim follows.

- (B-SETDOM) Then \( T = \{ \text{tab} \mapsto s \} \) with \( s = \text{setdom}(\text{be}, u, \bar{b}) \).

All properties of definition 11 except for property 3 are trivial, so we only show this one. We assume the following setting analogous to rule (B-SETDOM)

- \( P = P_0 \cup \{ \text{tab} \mapsto (u, f, l'', \mu'') \} \).
- \( \ell = \lambda(u) \).
- \( \text{eval}(\text{be}, M, f) = v' \).
- \( \forall k \in [1 \ldots |\text{be}|], v'_{k} = \text{eval}(\text{be}_{k}, M, f) \land v''_{k} = v''_{k} \sqcup \text{ref}_l \).
- \( \mu'' = (\mu = \text{att} \lor \mu'' = \text{att}) \lor \text{att} : \text{hon} \).

Then \( P' = P_0 \cup \{ \text{tab} \mapsto (u', f \langle v' \mapsto \text{form}(u \langle \mu \sqcup l \rangle, v^2) \rangle, l'' \sqcup l, \mu'' \} \). We now do a case analysis:

- \( I(\ell_a) \subseteq_I l'' \). Then by property 4 of definition 11 we know \( l = l'' \) and hence \( l \sqcup l = l'' \). We now need to show that with \( \Gamma_{\ell}(u) = \ell_{u}, \bar{a}_{u}, l_r \)
1) \( \Gamma_B(u) = \Gamma_X(v) \)
2) \( l \sqsubseteq I(\ell_a) \)
3) \( \forall i \in [1 \ldots |\vec{\ell}|], \tau_i \sqsubseteq I(\ell_a) \)
4) \( \mu'' = \text{hon} \)

(1) follows immediately from rule (T-BSetDOM).

From definition 11 we know by property 3 that \( l'' \sqsubseteq I(\ell_a) \) and by property 4 we know with \( \Gamma_B(u') = \ell_u^L, \tau_u^L, l_r^L \) that \( l'' = l_r \) and by rule (T-BSetDOM) we know that \( l'_x \sqsubseteq I(\ell_x) \) and (2) follows.

For (3), we get with rule (T-BSetDOM) and lemma 5 that \( \forall i \in [1 \ldots |\vec{\ell}|], \tau_i = \tau_{a_i} \) and the claim follows immediately.

(4) is trivial, since with \( I(\ell_a) \not\sqsubseteq l \) and \( I(\ell_a) \not\sqsubseteq l'' \) we also know \( \mu = \text{hon} \) and \( \mu'' = \text{hon} \).

- \( I(\ell_a) \sqsubseteq u' \). Then we need to show that on of the following holds
  * \( \forall i \in [1 \ldots |\vec{\ell}|], \tau_i \sqsubseteq \ell_a \ell_a \)
  * or \( I(\ell_a) \not\sqsubseteq u' \)

If \( I(\ell_a) \not\sqsubseteq u' \), the claim is trivial, we hence assume \( I(\ell_a) \sqsubseteq u' \). The claim then follows immediately from the observation that by rule (T-REPLY) scripts of low integrity URLs can never contain any values of high confidentiality.

We now show that Browsers remain well-typed if they send out and request and that every sent request is well-typed.

**Lemma 7 (Browser Request).** Whenever a browser \( B \xrightarrow{\alpha} B' \) with \( \alpha = \text{rec}(t_b; n, u, p, c, k, o) \cdot \mu \) and \( \Gamma \models \ell_a, \text{usr} B \). Then \( \Gamma \models \ell_a, \text{usr} \alpha \) and \( \Gamma \models \ell_a, \text{usr} B' \)

**Proof.** Let \( B = (N, M, P, T, Q, \vec{a})^{\text{ls}, \text{lt}, \mu} \) and We know that rule (B-FLUSH) is used. We hence have \( Q = \{ \alpha @ l' \} \) and \( B' = (N, M, P, T, \{ \}, \vec{a})^{\text{ls}, \text{lt}, \mu} \). \( \Gamma \models \ell_a, \text{usr} B' \) then follows immediately from \( \Gamma \models \ell_a, \text{usr} B \) We get \( \Gamma \models \ell_a, \text{usr} \alpha \) by property 5 of definition 11.

We now show that a well-typed browser receiving a well-typed response is still a well-typed browser. We have the additional assumptions that the integrity of the response is at most as high as the integrity of the browser and that either the attacked mode of the browser and the response are the same or that the response is attacked and the integrity of the responding URL is low.

**Lemma 8 (Browser Response).** Whenever a for a browser \( B = (N, M, P, T, Q, \vec{a})^{\text{ls}, \text{lt}, \mu} \) we have \( B \xrightarrow{\alpha} B' \) with \( \Gamma \models \ell_a, \text{usr} B, \alpha = \text{res}(t_b, n, u, u', c, k, \vec{v}, page, s)^{\text{lt'}, \mu'}, l \sqsubseteq l'' \) and \( \mu = \mu'' \lor \mu'' = \text{att} \land I(\ell_a) \sqsubseteq I(\lambda(u)) \) and \( \Gamma \models \ell_a, \text{usr} \alpha \) then \( \Gamma \models \ell_a, \text{usr} B' \).

**Proof.** \( B' = (N', M', P', T', Q', \vec{a})^{\text{ls}, \text{lt'}, \mu'} \) We show that \( B' \) fulfills the properties of definition 11. We know that the step \( \alpha \) was taken using rule (B-RECVLOAD) (B-RECVINCLUDE), or (B-REDIRECT). In all cases property 2 of definition 11 follows immediately from property 4 of definition 10. We now do a case distinction on the rule used.

- **(B-RECVLOAD)**
  - Property 1 follows immediately from property 3 of definition 10
  - For property 4 we do a case distinction:
    * if \( \mu' = \mu'' = \text{hon} \) then the claim follows from property 3 of definition 10
    * if \( \mu' = \text{att} \) then the claim follows from properties 5 and 6 of definition 10
  - Property 3 follows from property 5 of definition 10 for \( \mu' = \text{hon} \) and from 3 of definition 10 if \( \mu' = \text{att} \).
  - Property 5 is trivial.
  - Property 6 follows from the same property for \( B \), the \( nf \) on the tab for the page in \( B' \) is the same as the one for the network connection in \( B \).
  - Property 7 is trivial.

- **(B-RECVINCLUDE)**
  - Property 1 follows immediately from property 3 of definition 10 and property 1 of definition 11 for \( B \).
  - Property 5 is trivial.

- For property 4 we do a case distinction:
  * If \( \mu'' = \text{hon} \), then \( \mu = \mu' = \text{hon} \) and the claim follows from property 5 of definition 10 and property 4 of definition 11 for \( B \), using rule (T-BSEQ) and property 4 of definition 11.
  * if \( \mu' = \text{att} \) then \( \mu' = \text{att} \lor \mu = \text{att} \). Since we know that \( \mu = \text{att} \Rightarrow \mu' = \text{att} \) we can conclude that \( \mu' = \text{att} \). We distinguish two cases:
    - If \( \mu = \text{att} \) the claim follows immediately using property 3 of definition 10 and property 4 of definition 11 for \( B \), using rule (T-BSEQ).
\[ f \mu = \text{hon} \] then by the assumption in the lemma we have \( I(\ell_a) \sqsubseteq_I I(\lambda(u)) \) which is in contradiction to property \( \text{I} \) of definition \( \text{II} \) hence this case is impossible.

- Property \( \text{S} \) is trivial.
- Property \( \text{G} \) is trivial.
- Property \( \text{I} \) is trivial.

- (B-REDIR)

- Property \( \text{II} \) is trivial.
- Property \( \text{S} \) is trivial.
- Property \( \text{B} \) is trivial.

- For property \( \text{S} \) we know that \( Q' = \{ \alpha' \} \) with \( \alpha' = \overline{\text{EQ}}(i_b, n', u', p, c_k, o')^{\prime'} \) where
  
  * \( \forall k \in [1 \ldots \overline{\nu}] : p(k) = v_k \)
  * \( c_k' = \text{get}_c(M', u') \)
  * \( o' = (o = \text{orig}(u)) \ ? \ o : \perp \)

  and hence have to show that \( \Gamma \models_{i_b, \text{usr}} \alpha' \). We show that all the properties of definition \( \text{S} \) are fulfilled.

  - Property \( \text{I} \) follows immediately from property \( \text{X} \) of definition \( \text{S} \) for \( \alpha \)
  - Property \( \text{II} \) follows immediately from property \( \text{X} \) of definition \( \text{S} \) for \( \alpha \)
  - Property \( \text{S} \) follows immediately from property \( \text{II} \) of definition \( \text{II} \) for \( B \)
  - Property \( \text{B} \) is trivial since \( i_u = \text{usr} \)

  - For property \( \text{S} \) we perform a case distinction:
    
    * If \( u' \not\in P \), \( I(\ell_a) \sqsubseteq_I \alpha' \) or \( \mu' = \text{hon} \) then the claim is trivial.
    * If \( u \in P \), \( I(\ell_a) \not\subseteq_I \alpha' \) and \( \mu' = \text{att} \) then we know that \( I(\ell_a) \not\subseteq_I \text{orig}(u) \). We then know that the code at endpoint \( u \) can be typed with \( b = \text{hon} \) and we get by rule (T-REDIRECT) that \( u \not\in P \). Since the redirect URL must appear as a constant in the code, we apply this result in any case and reach a contradiction.

- Property \( \text{B} \) is trivial since high integrity pages only include high integrity pages.

- For property \( \text{B} \) we distinguish two cases:
  
  * If \( T = \{ \} \) or \( \mu' = \text{att} \) the claim is trivial
  * If \( T = \{ \} \) and \( \mu' = \text{hon} \), then we know that \( \mu = \text{hon} \) and \( \mu'' = \text{hon} \). The claim then follows using rule (T-REDIR)

\[ \square \]

We have now shown all lemmas for browser steps and move on to the server. First, we introduce typing for the server:

**Definition 12** (Server Typing). Let \( S = (D, \phi, t) \) be a server with \( D = (D_{SR}, D_S) \). We write \( \Gamma \models_{i_b, \text{usr}} S \), if

1. if \( i_b = \text{usr} \) then for all \( i \in \text{dom}(D_{SR}) \), for all \( r \in \text{dom}(D_{SR}(i)) \) we have if \( D_{SR}(i)(r) = v^\tau \) then \( \tau \sqsubseteq_{\ell_a} ref_r(\Gamma_{SR}(r)) \)
   
   - if \( i_b \neq \text{usr} \) then for all \( i \in \text{dom}(D_{SR}) \), for all \( r \in \text{dom}(D_{SR}(i)) \) we have if \( D_{SR}(i)(r) = v^\tau \) then \( \tau \sqsubseteq_{\ell_a} \ell_a \)

2. for all \( i \in \text{dom}(D_S) \), for all \( r \in \text{dom}(D_S(j)) \) we have if \( D_S(i)(r) = v^\tau \) then \( \tau = ref_r(\Gamma_{SR}(r)) \top \| \text{jlabel}(j) \)

3. for all \( u \in \text{urs}(S) \), for all \( j \in \text{dom}(\phi) \) we have that \( \text{ref}(\phi(j), u) \sqsubseteq_{\ell_a} j \text{label}(j) \)

4. \( \Gamma_{SR} \vdash_{i_b, \text{usr}} t \)

5. For all \( t \in \text{threads}(S) \) with \( t = \lceil c \rceil_{i_b, u, i_b, o, i, j} \) we have if \( i_b = \text{usr} \), \( u \in P \) and \( o \neq \perp \) and \( I(\ell_a) \not\subseteq_I o \) then \( \mu = \text{hon} \).

Intuitively, according to definition \( \text{I} \) a server is well typed if

1. For the global memories we have that
   
   * for honest users, all values respect the typing environment
   * for the attacker, all values are of the attackers \( \ell_a \)

2. All values in session memories respect the typing environment (taking the label of the session identifier into account)

3. All sessions are protected by session identifiers whose security guarantees are stronger than the one of the passwords corresponding to the identity stored in the session.

4. All server threads are well-typed.

5. For all threads the integrity label is as least as low as the origin

We now show the same lemmas we showed for the browser on the server side, starting with the substitution of variables in server expressions.

**Lemma 9** (Server Expression Substitution). Whenever we have \( \Gamma, \ell_a \vdash_{i_b} se : \tau \) and we have a substitution \( \sigma \) with \( \text{dom}(\sigma) = \text{dom}(\Gamma_X) \) and \( \forall x \in \text{dom}(\Gamma_X), \sigma(x) = v_x^{\sigma} \) with \( \tau_x \sqsubseteq_{\ell_a} \Gamma_X(x) \) then for all \( \Gamma' \), \( (\Gamma_U, \Gamma_X, \Gamma_{SR}, \Gamma_{SR}, \Gamma_Y), b \vdash_{\ell_a} se : \tau \).

**Proof.** We perform an induction on the typing derivation of \( \Gamma, \ell_a \vdash_{i_b} se : \tau \):

- (T-EVAR). Then \( se = x \) and \( v_x^{\sigma} \sqsubseteq_{\ell_a} \Gamma_X(x) \). The claim follows directly from rule (T-EVAL) and (T-SUB).
- (T-ESesRef). Then se = @r = sen and the claim is trivial.
- (T-EglobRef). Then se = $r = sen and the claim is trivial.
- (T-Eval). Then se = \textit{v}^\tau = sen and the claim is trivial.
- (T-Undef). Then se = \bot = sen and the claim is trivial.
- (T-Ename). Then se = n^\tau = sen and the claim is trivial.
- (T-ES). Then se = fresh() = sen and the claim is trivial.
- (T-Binop) Then se = se_1 \odot se_2 with \Gamma, \ell_s \vdash^\tau se_1 : \tau_1 and \Gamma, \ell_s \vdash^se_2 : \tau_2 and \tau = label(\tau_1) \sqcup label(\tau_2). We also have sen = se_1 sen \odot sen_2. By induction we know that \Gamma, \ell_s \vdash^se_1 sen_1 : \tau_1' and \Gamma, \ell_s \vdash^se_2 sen_2 : \tau_2' with \tau_1' \sqsubseteq \ell_s \tau_1 and \tau_2' \sqsubseteq \ell_s \tau_2. Then we know that label(\tau_1') \sqcup label(\tau_2') \subseteq \ell_s label(\tau_1) \sqcup label(\tau_2) = \tau, and the claim follows by (T-Binop) and (T-Besub).
- (T-ESub) follows by induction and by the transitivity of \sqsubseteq_{\ell_s}.

To show the substitution lemma for server commands we first need to show auxiliary lemmas that deal with the program counter.

First, we show that whenever server code can be typed with a pc, it can also be typed with any pc of higher integrity.

**Lemma 10** (Server Program Counter Substitution). Whenever we have \Gamma, \ell_s, pc \vdash^\varepsilon \ell_{a,(u,b,p)} c : \ell_s', pc' and pc* \sqsubseteq I pc then \Gamma, \ell_s, pc' \vdash^\varepsilon \ell_{a,(u,b,p)} c' : \ell_s'', pc'' with pc''* \sqsubseteq I pc'.

**Proof.** We perform the proof by induction on the typing derivation.
- (T-Skip) The claim is trivial.
- (T-Login) The claim follows from the transitivity of \sqsubseteq I.
- (T-Start) The claim is trivial.
- (T-Setglobal) The claim follows from the transitivity of \sqsubseteq I.
- (T-SetSession) The claim follows from the transitivity of \sqsubseteq I.
- (T-Seq) The claim follows by induction on the two subcommands.
- (T-Iif) Then c = if se then c_1 else c_2 with
  - \Gamma, \ell_s \vdash^se : \tau.
  - pc' = pc \sqcup I (\tau).
  - \Gamma, \ell_s, pc'' \vdash^\varepsilon \ell_{a,(u,b,p)} c_1 : \ell_s'', pc_1.
  - \Gamma, \ell_s, pc'' \vdash^\varepsilon \ell_{a,(u,b,p)} c_2 : \ell_s'', pc_2.
  - pc'' = ((c and c' do not contain reply, redir, tokencheck or origincheck) ? pc : pc') \sqcup I pc_1 \sqcup I pc_2.
Let pc'' = pc''* \sqsubseteq I pc' and we can apply the induction hypothesis for c_1 and c_2 and get
  - \Gamma, \ell_s, pc'' \vdash^\varepsilon \ell_{a,(u,b,p)} c_1 : \ell_s'', pc_1.
  - \Gamma, \ell_s, pc'' \vdash^\varepsilon \ell_{a,(u,b,p)} c_2 : \ell_s'', pc_2.
with pc_1* \sqsubseteq I pc_1 and pc_2* \sqsubseteq I pc_2 and the claim follows by applying (T-Iif).
- (T-Tcheck) The claim is trivial.
- (T-PruneTcheck) The claim is trivial.
- (T-Ohchck) The claim is trivial.
- (T-PruneOhchck) The claim is trivial.
- (T-Reply) With \Gamma_I(u) = \ell_u, \ell_l, \ell_r, we let pc' = pc \sqcup I l_r and pc'' = pc \sqcup I l_r. We do a case distinction on b:
  - If b = hon we get pc \sqsubseteq I l_r, hence pc' = l_r and because of of pc' \sqsubseteq I pc we also get pc'' = l_r and the claim follows.
  - If b \neq hon we have pc = \top and we hence also have pc* = \top and the claim follows.
- (T-Redirect) The claim follows from the transitivity of \sqsubseteq I.
- (T-Reset) The claim is trivial.

We now show that if server code containing variables is well typed in a typing environment typing these variables, then the code is also well typed after instantiating these variables with concrete values of the same type.

**Lemma 11** (Server Substitution). Whenever we have \Gamma, \ell_s, pc \vdash^\varepsilon \ell_{a,(u,b,p)} c : \ell_s', pc* and we have a substitution \sigma with \text{dom}(\sigma) = \text{dom}(\Gamma_X) and \forall x \in \text{dom}(\Gamma_X). \sigma(x) = \textit{v}_x^\tau with \tau_x \sqsubseteq \ell_s \Gamma_X(x) then for all \Gamma_X, we have \Gamma', \ell_s, pc \vdash^\varepsilon \ell_{a,(u,b,p)} c_σ : \ell_s', pc_{\sigma'*} with \Gamma' = (\Gamma_U, \Gamma_X, \Gamma_R*, \Gamma_R^*, \Gamma_V).

**Proof.** We do the proof by induction on the typing derivation.
• (T-BSEQ): Then \( c = c_1, c_2 \). The claim follows by applying the induction hypothesis to \( s_1 \) and \( s_2 \) using lemma 10 and applying rule (T-SEQ)
• (T-SKIP): The claim follows trivially.
• (T-SETHOOKSESSION): The claim follows from lemma 9 and the transitivity of \( \sqsubseteq_{\ell_a} \).
• (T-SETGLOBAL): The claim follows from lemma 9 and the transitivity of \( \sqsubseteq_{\ell_a} \).
• (T-LOGIN): The claim follows from lemma 9 and the transitivity of \( \sqsubseteq_{\ell_a} \).
• (T-START): The claim follows from lemma 9 and the transitivity of \( \sqsubseteq_{\ell_a} \).
• (T-IF): Then \( c = \text{if } se \text{ then } c_1 \text{ else } c_2 \) with
  - \( \Gamma, \ell_s \vdash_{\ell_a} se : \tau \)
  - \( pC' = pC \cup I(\tau) \)
  - \( \Gamma, \ell_s, pC' \vdash_{\ell_a} (u,b,P) c_1 : \ell''_{s}, pC_1 \)
  - \( \Gamma, \ell_s, pC' \vdash_{\ell_a} (u,b,P) c_2 : \ell''_{s}, pC_2 \)

  By induction we know
  - \( \Gamma', \ell_s, pC' \vdash_{\ell_a} (u,b,P) c_1 \sigma : \ell''_{s}, pC_1^\ast \)
  - \( \Gamma', \ell_s, pC' \vdash_{\ell_a} (u,b,P) c_2 \sigma : \ell''_{s}, pC_2^\ast \)

  By lemma 9 we know that \( \Gamma', \ell_s \vdash_{\ell_a} \exists \sigma \tau \). The claim then follows by applying rule (T-IF).
• (T-AUTH) The claim follows from lemma 9 and the transitivity of \( \sqsubseteq_{\ell_a} \).
• (T-PRUNEHECK) The claim follows from lemma 9 and the fact that there is no subtyping on credentials of high confidentiality.
• (T-OCHKCSUCC) The claim follows trivially.
• (T-OCHKFAIL) The claim follows trivially.
• (T-HECK) The claim follows from lemma 9.
• (T-REPLY) Let variables be assigned as in the rule. The claim then follows by applying lemma 9 for all \( se_k \). The claim then follows immediately.
• (T-REDIR) Let variables be assigned as in the rule. The claim then follows by applying lemma 9 for all \( se_k \).

We now show that typing of server expressions is preserved under evaluation.

Lemma 12 (Server Expression Typing). Let \( S = (D, \phi, t) \) be a server with \( \Gamma \models_{\ell_a, \text{var}} S \) and let \([c]_{R,i,j}^{l,\mu} \in \text{running}(S)\). Then for any server expression \( se \), if \( \Gamma, jlabel(j) \vdash_{\ell_a} se \) then \( \Gamma, jlabel(j) \vdash_{\ell_a} evali,j(se, D) : \tau \).

Proof. By induction over the expression \( se \).
  - \( se = v^\tau \): Then \( evali,j(se, D) = se \) and the claim is trivial.
  - \( se = se_1 \circ se_2 \): By induction analog to case in in lemma 5.
  - \( se = \# \): straightforward from property 1 of definition 12 using (T-EGLOBRef) and (T-SUB).
  - \( se = \$ \): straightforward from property 2 of definition 12 using (T-ESResRef) and (T-SUB).
  - \( se =\text{fresh}()^\tau \): straightforward from (SE-FRESH), (T-FRESH) and (T-ENAME).

Next, we show that whenever a server thread is typable with the session label \( \times \), then it is also typable with any other session label.

Lemma 13 (Server Typing with \( \ell_s = \times \)). Whenever we have \( \Gamma, \times, pC \vdash_{\ell_a}\ c : \ell''_{s}, pC \) then we also have \( \Gamma, \ell_s, pC \vdash_{\ell_a}(u,b,P) c : \ell''_{s}, pC \) for all \( \ell_s \), where \( \ell_s = \times \) or \( \ell_s = \ell''_{s} \).

Proof. This is simple by inspecting the typing rules and the observation that \( \ell_s = \times \) implies that the session memory cannot be used. Hence the code that is typed with \( \ell_s = \times \) can be typed with any session label. Furthermore, if the session label is set to a different label during typing, this is unaffected by the old session label.

We are now ready to show that whenever a well-typed server takes an internal step, it results in another well-typed server.

Lemma 14 (Server Subject Reduction). Let \( S \) be a server with \( \Gamma^0 \models_{\ell_a, \text{var}} S \) and \( S \overset{\alpha}{\rightarrow} S' \), where \( \alpha \in \Sigma^\ell_{v \in \ell_a} \). Then we have \( \Gamma^0 \models_{\ell_a, \text{var}} S' \).

Proof. Let \( S = (D, \phi, t) \) and let \( S' = (D', \phi', t') \). Then there exists \([c]_{R,i,j}^{l,\mu} \in \text{running}(S)\) with \((D, \phi, [c]_{R,i,j}^{l,\mu}) \overset{\alpha}{\rightarrow} (D', \phi', [c']_{R,i,j}^{l,\mu})\).
Because of rules (S-Parallel), (S-RParallel) and (T-Parallel) it is sufficient to show \( \Gamma^0 \models_{\epsilon, \text{usr}} (D, \phi, [c']^{l''\mu}_{R, i, j'}) \), assuming \( \Gamma^0 \models_{\epsilon, \text{usr}} (D, \phi, [c]^{l\mu}_{R, i, j}) \).

We chose \( b, \ell_s, \text{pc}_1 \) and \( \Gamma \) as in rule (T-Running):

\[
\begin{cases}
\text{hon} & \text{if } \mu = \text{hon} \land t_b = \text{usr} \\
\text{csrf} & \text{if } \mu = \text{att} \land t_b = \text{usr} \text{ and } \ell_s = jlabel(j) \text{ and let } \text{pc}_1 = l. \\
\text{att} & \text{if } t_b \neq \text{usr}
\end{cases}
\]

With \( \Gamma^0 = (\Gamma, \Gamma_X, \Gamma_{R^a}, \Gamma_{R^b}, \Gamma_Y) \) and \( \Gamma_{R^a} = (t_b = \text{usr}) \) ? \( \Gamma_{R^a} = \{ \rightarrow \ell_a \} \) we let \( \Gamma = (\Gamma_U, \Gamma_X, \Gamma_{R^a}, \Gamma_{R^b}, \Gamma_Y) \).

We furthermore let \( \ell_s = jlabel(j') \) and \( \text{pc}_2 = l' \).

We now show that \( S' \) fulfills all properties of definition (12).

However, for property (5) of definition (12) we will show the following stronger claim:

Whenever \( \Gamma, \ell_s, \text{pc}_1 \models_{\epsilon, (u,b,p)} c : \ell_s, \text{pc}_1' \) we have \( \Gamma, \ell_s, \text{pc}_2 \models_{\epsilon, (u,b,p)} c' : \ell_s, \text{pc}_2' \) where \( \text{pc}_2 \models_{I} \text{pc}_1' \) and \( \ell_s = \ell_s' \)

For all cases property (5) is trivial.

We perform the proof by induction the step taken.

- (S-Skip). This case is trivial.
- (S-SEQ) Then we know
  \[
  - c = \ell_s; \text{c}_2
  - (D, \phi, [c]^{l\mu}_{R, i, j}) \Rightarrow (D', \phi', [c']^{l''\mu}_{R, i, j})
  - c' = \ell_s; \text{c}_2.
  \]

All properties of definition (12) except for property (4) follow immediately by the induction hypothesis applied to \( c_1 \).

By rule (T-SEQ) we know that for some \( \text{pc}_1'' \) and \( \ell_s'' \)

\[
\begin{align*}
\Gamma, \ell_s, \text{pc}_1' & \models_{\epsilon, C} \ell_s'' : \ell_s', \text{pc}_1'' \\
\Gamma, \ell_s'', \text{pc}_2' & \models_{\epsilon, C} \ell_s', \text{pc}_1''
\end{align*}
\]

By induction we know that for some \( \text{pc}_2, \text{pc}_2', \ell_s, \ell_s'' \)

\[
\begin{align*}
\Gamma, \ell_s, \text{pc}_2, \text{pc}_2' & \models_{\epsilon, (u,b,p)} \ell_s' : \ell_s', \text{pc}_2'' \\
\text{pc}_2'' & \models_{I} \text{pc}_1'' \\
\ell_s'' & = \times \text{ or } \ell_s'' = \ell_s'
\end{align*}
\]

Using lemma (10) we get \( \Gamma, \ell_s'', \text{pc}_2' \models_{\epsilon, C} \ell_s'' : \ell_s', \text{pc}_2'' \) with \( \text{pc}_2'' \models_{I} \text{pc}_1'' \)

Using lemma (13) we furthermore get \( \Gamma, \ell_s', \text{pc}_2, \text{pc}_2' \models_{\epsilon, C} \ell_s', \text{pc}_2'' \) with \( \ell_s' = \times \) or \( \ell_s' = \ell_s'' \)

Using (T-SEQ) we can then conclude \( \Gamma, \ell_s, \text{pc}_2, \text{pc}_2' \models_{\epsilon, C} \ell_s', \text{pc}_2'' \) and the claim follows.

- (S-IfTrue) then
  \[
  \begin{align*}
  c & = \text{if } \text{se} \text{ then } \text{c}_1 \text{ else } \text{c}_2 \\
  \text{eval}_{i,j}(\text{se}, D) & = \text{true}^\tau \\
  j' & = j \\
  l' & = l \cup I(\tau) \\
  c' & = (\text{reply}, \text{redir}, \text{tokencheck}, \text{origincheck} \in \text{coms}(c)) \text{ ? } \text{c}_1 : c; \text{reset } \text{l}
  \end{align*}
  \]

All properties of definition (12) except for property (4) follow immediately by the induction hypothesis applied to \( c_1 \) using rules (S-SEQ) and (S-Reset).

By (T-If) we know

\[
\begin{align*}
\Gamma, \ell_s & \models_{\epsilon, C} \text{se} : \tau' \\
\text{pc}_c & = \text{pc}_1 \cup I(\tau) \\
\Gamma, \ell_s, \text{pc}_c' & \models_{\epsilon, C} \ell_s' : \ell_s', \text{pc}_c' \text{ for some } \ell_s', \text{pc}_c' \\
\Gamma, \ell_s, \text{pc}_c' & \models_{\epsilon, C} \ell_s' : \ell_s', \text{pc}_c'' \text{ for some } \ell_s'', \text{pc}_c'' \\
\ell_s' & = \ell_s' = \ell_s'' = \times \\
\text{pc}_c' & = \text{reply}, \text{redir}, \text{tokencheck}, \text{origincheck} \in \text{coms}(c) \text{ ? } \text{pc}_2' \cup l \text{ \pc}_c'' : \text{pc}_1
  \end{align*}
\]

By lemma (12) we know that \( \tau \models_{\epsilon} \tau' \). Hence \( \text{pc}_2 = l' \models_\tau \text{pc}_c' \). We thus have by lemma (10) that \( \Gamma, \ell_s, \text{pc}_2 \models_{\epsilon, C} \ell_s' : \ell_s', \text{pc}_2'' \) with \( \text{pc}_2'' \models_{I} \text{pc}_c' \).

If reply, redir, tokencheck, origincheck \in \text{coms}(c), the claim follows immediately.

Otherwise, using (T-SEQ) and (T-Reset) we observe that \( \Gamma, \ell_s, \text{pc}_2 \models_{\epsilon, C} \text{c}_1; \text{reset } l : \ell_s', l \). With \( \text{pc}_c' = \text{pc}_1 = l = \text{pc}_2'' \) the claim follows immediately.

- (S-IfFalse) then the claim follows analog to the previous one.
- (S-TcTrue) Then
  \[
  c = \text{if } \text{tokenchk}(\text{se}, \text{se}') \text{ then } \text{c}'
  \]
- \( eval_{i,j}(se, D) = v_1^{\tau_1} \)
- \( eval_{i,j}(se', D) = v_1^{\tau_2} \)
- \( v_1 = v_2 \)
- \( l' = l \)
- \( j' = j \)

All properties of definition 12 except for property 4 are trivial.

We know that typing was done using rule \( (T\text{-TC}) \) or \( (T\text{-PRUNE}) \).

We want to show that \( (T\text{-TC}) \) was used. To this end, we assume that \( (T\text{-PRUNE}) \) was used and show a contradiction.

By rule \( (T\text{-TC}) \) we know
- \( se = x \) for some \( x \) and \( \Gamma, \ell_s \vdash^e x : \tau_1' \)
- \( se' = r \) for some \( r \) and \( \Gamma, \ell_s \vdash^e r : \tau_2' \)
- \( \tau_2 = \text{cred}(\ell) \)
- \( \tau_1 \neq \text{cred}(\ell) \)
- \( \text{cred}(\ell) \not\sqsubseteq C(\ell_a) \)

By lemma 12 we know that \( \tau_1 \sqsubseteq_{\ell_a} \tau_1' \) and \( \tau_2 \sqsubseteq_{\ell_a} \tau_2' \). By the definition of \( \sqsubseteq_{\ell_a} \), we know that \( \tau_1 = \tau_1' \). Since the set of credentials at label \( \text{label}(\tau_1) \) is disjoint from the set of the set of any other values, and since \( v_1 = v_2 \), we know that also \( \tau_1 = \tau_2 \).

Using the definition of \( \sqsubseteq_{\ell_a} \) we get \( \tau_2 = \tau_2' \). We hence have \( \tau_1' = \tau_2' \) which contradicts the assumption.

We thus know that \( (T\text{-TC}) \) and we get \( \Gamma, \ell_{s_1}, p_c \vdash^e_{\ell_{s_1}(u,b), p} c' : \ell'_{s_1}, p_c' \) and the claim follows.

- \( (S\text{-TCfalse}) \) Then
- \( c = \text{if tokencheck}(se, se') \text{ then } c'' \)
- \( c' = \text{reply} \{ \text{error, skip}, \{ \} \} \)
- \( \text{eval}_{i,j}(se, D) = v_1^{\tau_1} \)
- \( \text{eval}_{i,j}(se', D) = v_1^{\tau_2} \)
- \( v_1 \neq v_2 \)
- \( l' = l \)
- \( j' = j \)

All properties of definition 12 except for property 4 are trivial.

By \( (T\text{-reply}) \) we immediately get \( \Gamma, \ell_{s_1}, p_c \vdash^e_{\ell_{s_1}(u,b), p} \text{reply} \{ \text{error, skip}, \{ \} \} : \ell_1, p_c. \)

- \( (S\text{-Reset}) \) All properties of definition 12 are trivial, where property 4 follows immediately from \( (T\text{-Reset}) \).

- \( (S\text{-RestoreSession}) \) We have
- \( c = \text{start se} \)
- \( c' = \text{skip} \)
- \( \text{eval}_{i,j}(se, D) = v^\tau \)
- \( v \in \text{dom}(D_s) \)
- \( l' = C(\tau) \sqsubseteq C(\ell_a) \) \( \text{val} \) \( : \text{label}(\tau) \)
- \( j' = v \)

All properties of definition 12 except for property 4 are trivial.

By \( (T\text{-Start}) \) we get
- \( \Gamma, \ell_s \vdash^e se : \text{cred}(\ell) \)
- \( \ell_{s_1}' = (\overline{C(\text{cred}(\ell)) \sqsubseteq C(\ell_a)} : (\bot_C, \top_I) : \ell) \)

By lemma 12 we know that \( \tau \sqsubseteq_{\ell_a} \text{cred}(\ell) \) \( \ell \). We distinguish two cases:
- If \( C(\text{cred}(\ell)) \not\sqsubseteq C(\ell_a) \) then also \( C(\tau) \not\sqsubseteq C(\ell_a) \) and we have \( \ell_{s_1}' = (\bot_C, \top_I) = j\text{label}(j') = \ell_{s_2}' \).
- If \( C(\text{cred}(\ell)) \not\sqsubseteq C(\ell_a) \) we know \( \tau = \text{cred}(\ell) \) and we have \( \ell_{s_1}' = \ell = j\text{label}(j') = \ell_{s_2}' \).

- \( (S\text{-NewSession}) \) We immediately get property 13 of definition 12 Property 2 follows immediately using rule \( (T\text{-EUNDEV}) \) since the freshly created memory is empty. Property 4 follows analog to the previous case.

- \( (S\text{-SetGlobal}) \) We immediately get property 234 of definition 12 using \( (T\text{-SetGlobal}) \).

We have \( c = @r := se, \text{eval}_{i,j}(se, D) = v^\tau \) and \( \tau' = \tau \sqcup \text{ref}_r(\Gamma'_{R_o}(r)) \sqcup l \) with \( \Gamma'_{R_o} = (t_b = \text{usr}) \) \( \Gamma'_{R_o} : \{ \_ \rightarrow \ell_a \} \).

We know using rule \( (T\text{-SetGlobal}) \) that
- \( \Gamma', @r \vdash^e \text{ref}(\tau') \)
- \( \Gamma', \ell_{s_1} \vdash^e se : \tau' \)
- \( p_c \vdash^e \ell_I(\tau') \)

Using \( (T\text{-GlobRef}) \) and \( (T\text{-RefSub}) \) we know that \( \tau' \sqsubseteq_{\ell_a} \text{ref}_r(\Gamma'_{R_o}(r)) \).

Using lemma 12 we know \( \tau \sqsubseteq_{\ell_a} \tau' \).

We hence get that \( \tau' = \text{ref}_r(\Gamma'_{R_o}(r)) \) and property 1 follows.

- \( (S\text{-SetSession}) \) We immediately get property 134 of definition 12 using rule \( (T\text{-SetSession}) \).
We have $c = \$r := se$, $eval_{\ell,j}(se, D) = v^\tau$ and $\tau' = \tau \sqcup (\text{ref}_r(\Gamma_{R^s}(r)) \vec{j} \text{label}(j)) \sqcup I_l$.

We know using rule (T-SetSession) that

- $\Gamma, \$r \vdash^P_{\ell_a} \text{ref}(\tau')$
- $\Gamma, \ell \vdash^\ell_a se : \tau'$
- $p \in C \sqsubseteq I_l(\tau')$

Using (T-SetRef) and (T-RefSub) we know that $\tau' \sqsubseteq \ell_a (\text{ref}_r(\Gamma_{R^s}(r)) \vec{j} \text{label}(j))$. Using lemma 12 we know $\tau \sqsubseteq \ell_a \tau'$. We hence know that $\tau' = \text{ref}_r(\Gamma_{R^s}(r)) \vec{j} \text{label}(j)$ and the property 2 follows.

- (S-LOGIN) We immediately get property 12 of definition 12. Property 3 follows from rule (T-Login).
- (S-AUTH) All properties are trivial.
- (S-OKCKSUCC) Then
  - $c = \text{if originchk}(L) \text{ then } c'$
  - $R = n, u, t_b, o$
  - $o \in L$

All properties of definition 12 except for property 4 are trivial.

We know that typing was done using rule (T-OCHK) or (T-PruneOCHK).

We want to show that (T-OCHK) was used. To this end, we assume that (T-PruneOCHK) was used and show a contradiction.

By rule (T-OCKKPRUNE) we know

- $\forall l \in L. I(\ell_a) \not\subseteq I_o$
- $u \in P$
- $b = \text{csrf}$

We hence have $I(\ell_a) \not\subseteq I_o$.

Then by 5 of definition 12 we know that $\mu = \text{hon}$, which is an immediate contradiction.

- (T-OCKFFAIL) This case is analogous to the case of rule (T-TOCKFAIL)

We now show that any expression that is well-typed in an honest typing branch is also well-typed when typing in the attacker’s setting and that all expressions have type $\ell_a$ in the attacked setting.

**Lemma 15 (Attacker Server Expression Typability).** For all server expressions $se$ we have if

- $\Gamma, \ell_a \vdash^\ell_a se : \tau$
- $\forall x \in x, \Gamma_{\ell_a}(x) = \ell_a$
- $\forall r \in R. \Gamma_{R^a}(r) = \text{ref}(\ell_a)$
- $\ell_a \not\times \Rightarrow \ell_a' = \ell_a$
- $se = \overline{se'}$ for some $se'$

then we have $(\Gamma_{\ell_l}, \Gamma_X, \Gamma_{R^a}, \Gamma_{R^s}, \Gamma_Y), \ell_a' \vdash^\ell_a se : \ell_a$

**Proof.** We prove the claim by induction over the typing derivation for $\Gamma, \ell_a \vdash^\ell_a se : \tau$

- (T-EVAL) Since $se = \overline{se'}$ from some $se'$, we have $se \equiv v^\tau$. The claim then follows since $\bot \sqsubseteq \ell_a \ell_a$.
- (T-EFRESH) Then we have $se = \text{fresh}(\ell)$. The claim is trivial because of $b = \text{att}$.
- (T-VUNDEF) Trivial.
- (T-EVAR) Follows immediately from the definition of $\Gamma_{X'}$.
- (T-EGLOBREF) Follows immediately from the definition of $\Gamma_{R^a}$.
- (T-ESERSREF) Then we know that $\ell_a \not\times$ and hence $\ell_a' = \ell_a$. The claim then follows immediately from (T-ESERSREF) and (T-ESUB).
- (T-EBINOP) Then the claim follows immediately by induction.
- (T-ESUB) The claim follows immediately by induction.

Next we show, that any server thread that is well-typed in the honest setting is also well-typed when typing in the attacker’s setting.

**Lemma 16 (Attacker Server Typability).** Let $t$ be a thread with

- $t = u[\vec{r}](\vec{x}) \rightarrow c$ with $\Gamma^0 \vdash^\ell_{a,p} t$
- $\forall x \in x, \Gamma_X(x) = \ell_a$
- $\forall r \in R. \Gamma_{R^a}(r) = \ell_a$
- $t = \vec{r}$ for some $t'$

we have $t \vdash^\ell_{a,p} t$.

**Proof.** We prove the claim by induction on the typing derivation for $t \vdash^\ell_{a,p} t$

- (T-EVAL) Since $t \equiv v^\ell$. The claim then follows.
- (T-EFRESH) Then we have $t = \text{fresh}(\ell)$. The claim is trivial because of $b = \text{att}$.
- (T-VUNDEF) Trivial.
- (T-EVAR) Follows immediately from the definition of $\Gamma_{X'}$.
- (T-EGLOBREF) Follows immediately from the definition of $\Gamma_{R^a}$.
- (T-ESERSREF) Then we know that $\ell_a \not\times$ and hence $\ell_a' = \ell_a$. The claim then follows immediately from (T-ESERSREF) and (T-ESUB).
- (T-EBINOP) Then the claim follows immediately by induction.
- (T-ESUB) The claim follows immediately by induction.
we have $(\Gamma_0^0, \Gamma_X, \Gamma_{R,s}, \Gamma_{R,t}, \Gamma_Y^0), \times, T_I \models_{\ell_a, (u, att, p)} c : \omega$.

**Proof.** By $\Gamma^0 \vdash_{\ell_a, p} t$ we know by (T-RECV) that with $\Gamma_{ua}(u) = \ell_u, r, l_r$ and $m = |x|$ and $\Gamma_0^h = x_1, \ldots, x_m : \tau_m$ we have

$$(\Gamma_0^0, \Gamma_X, \Gamma_{R,s}, \Gamma_{R,t}, \Gamma_Y^0), \times, I(\ell_u) \models_{\ell_a, (u, hon, p)} c : \omega$$

We let $\Gamma = (\Gamma_0^0, \Gamma_X, \Gamma_{R,s}, \Gamma_{R,t}, \Gamma_Y^0)$ and now show the following stronger claim: Whenever

$$(\Gamma_0^0, \Gamma_X, \Gamma_{R,s}, \Gamma_{R,t}, \Gamma_Y^0), \ell_{sh}, p \sigma_h \models_{\ell_a, (u, hon, p)} c : \ell_{sh}', p \sigma_h'$$

then

$$\Gamma, \ell_s, T_I \models_{\ell_a, (u, att, p)} c : \ell_{s}', \ell_I$$

where

- $\ell_{sh} = x \Rightarrow \ell_s = \times \land \ell_{sh} \neq x \Rightarrow \ell_s = \ell_a$ and
- $\ell_{sh}' = x \Rightarrow \ell_{s}' = \times \land \ell_{sh}' \neq x \Rightarrow \ell_{s}' = \ell_a$ and

The proof is by induction on the honest typing derivation for $c$.

- **(T-Skip)** The claim is trivial.
- **(T-Seq)** The claim follows directly from the induction hypothesis on the two subcommands.
- **(T-If)** We have $c = \text{if } s e \text{ then } c_t \text{ else } c_f$. With $\Gamma, \ell_s \models_{\ell_a} s : \ell_s$ by lemma 15. We have $p c' = T_I \cup_I I(\ell_s) = T_I = p c$ (in rule (T-IF)) and the claim follows immediately using (T-LOGIN).
- **(T-Login)** By lemma 16 we get that all expressions are of type $\ell_a$. Using rule (T-ESUB) we can also treat them as expressions of type $\mathsf{cred}(\bot_C, T_I)$, and the claim then follows immediately using (T-LOGIN).
- **(T-Start)** By lemma 16 we get that all expressions are of type $\ell_a$. Using rule (T-ESUB) we can also treat them as expressions of type $\mathsf{cred}(\bot_C, T_I)$, and the claim then follows immediately using (T-START).
- **(T-SetGlobal)** We have $c = \text{if } \tau : = e \text{ with } \Gamma, \ell_s \models_{\ell_a} : \ell_s$ by lemma 15 and $\Gamma_{R,s}(\tau) = \text{ref}(\ell_s)$. Using subtyping we can show $\Gamma, \ell_s \models_{\ell_a} \tau : \ell_a$. Using rule (T-SETGLOBAL) we get the following stronger claim: Whenever

$$(\Gamma_0^0, \Gamma_X, \Gamma_{R,s}, \Gamma_{R,t}, \Gamma_Y^0), \ell_{sh}, p \sigma_h \models_{\ell_a, (u, hon, p)} c : \ell_{sh}', p \sigma_h'$$

then

$$\Gamma, \ell_s, T_I \models_{\ell_a, (u, att, p)} c : \ell_{s}', \ell_I$$

- **(T-PruneTCheck):** Impossible since this rule cannot be applied for $b = \text{hon}$.
- **(T-TokenCheck):** By lemma 16 we get that all expressions are of type $\ell_a$. Using rule (T-ESUB) we can also treat them as expressions of type $\mathsf{cred}(\bot_C, T_I)$.
- **(T-PruneOCHK)** Impossible since this rule cannot be applied for $b = \text{hon}$.
- **(T-OCHK)** The claim follows immediately by induction.
- **(T-Auth):** Then the claim follows immediately using rule (T-AUTHATT).
- **(T-Reply)** From lemma 16 and rule (T-ESUB) we know that for all variables $x$ in the freshly generated environment $\Gamma' = (\bot_C, T_I)$. Furthermore, with subtyping we can show $\Gamma, \ell_s \models_{\ell_a} \tau : \text{ref}(\bot_C, T_I)$ for all $\tau \in \text{dom}(c_k)$. The claim then follows immediately.
- **(T-Redir)** This case follows analogous to the previous case.

Next we show that whenever a server receives a well typed request, the resulting running thread is also well-typed.

**Lemma 17** (Server Request). *Whenever a server $S \xrightarrow{\alpha} S'$ with $\Gamma \models_{\ell_a, \text{star}} S, \alpha = \text{req}(t_b, n, u, p, c_k, a)$ and $\Gamma \models_{\ell_a, \text{star}} \alpha$ then $\Gamma \models_{\ell_a, \text{star}} S'$***

**Proof.** Let $S = (D, \phi, t)$ and $S' = (D', \phi', t')$. We show that $S'$ fulfills the properties of definition 12. Property 2 and 3 follow immediately from rule (S-RECV) since the session memory and the trust mapping do not change.

- For property 1 we perform a case distinction
  - if $t_b \neq \text{usr}$ then property 1 follows from property 4 of definition 9.
  - if $t_b = \text{usr}$ then property 1 follows from property 3 of definition 9.

- For property 4, because of $S \xrightarrow{a} S'$ we know $(D, \phi, u[\tau]^\ell(\bar{x}) \rightarrow c) \xrightarrow{a} (D', \phi', [c]_{\ell_a}^{\ell_a} | u[\tau]^\ell(\bar{x}) \rightarrow c)$. It is hence sufficient, because of rule (T-Parallel), to show $\Gamma^0 \models_{\ell_a, p} [c]_{\ell_a}^{\ell_a} | u[\tau]^\ell(\bar{x}) \rightarrow c)$. We perform a case distinction:
  - if $t_b \neq \text{usr}$ then by rule (T-Running) with $b = \text{att}$ and $\ell_s = j \text{label}(\bot) = x$, $\Gamma_{R,s} = \{ x \mapsto \ell_a \}$ we have to show $$(\Gamma_U, \Gamma_X, \Gamma_{R,s}, \Gamma_Y, x, \ell_s, l \models_{\ell_a} (u, b, p) c \sigma : \ell_s', l)$$

Because of lemma 16 we get with $\Gamma' = x_1, \ldots, x_m : \ell_a$
We can now define the typing of websystems, which simply states that all browsers and servers contained in the system are well typed.

**Definition 13 (System Typing).** Let \( W \) be a websystem. We write \( \Gamma \vdash_{\ell_a,usr} (\ell_a, \mathcal{K}) \triangleright_{TO} W \), if

1) for all \( S \in \text{servers}(W) \) we have \( \Gamma \vdash_{\ell_a,usr} S \)
2) for all \( B \in \text{browsers}(W) \) we have \( \Gamma \vdash_{\ell_a,usr} B \)
3) for all \( B \in \text{browsers}(W) \) with \( B = (N,M,P,T,Q,\overline{a})^{i,j,\mu} \) and \( N = \{n \mapsto u\} \) we have one of the following:
   - there exists \( S \in \text{servers}(W) \) with \( t \in \text{running}(S) \), \( t = [e]^{i,j,\mu}_{(n,u,i,b),(i,j)} \) and \( l \subseteq \text{int}(t) \) for some \( c,l',i,j \),
   - or \( l(\ell_a) \subseteq \lambda(u) \)

We now show that all responses by the server fulfill these conditions.

**Lemma 18 (Server Response).** Whenever a server \( S \xrightarrow{\alpha} S' \) with \( \Gamma \vdash_{\ell_a,usr} S \), \( \alpha = \text{res}(t_d, n, u', \overline{v}, \overline{c}, \overline{k}, \overline{p}, s)^{i,j,\mu} \) then \( \Gamma \vdash_{\ell_a,usr} \alpha \) and \( \Gamma \vdash_{\ell_a,usr} S' \)

**Proof.** \( \Gamma \vdash_{\ell_a,usr} S' \) is trivial in all cases. We show that \( \alpha \) fulfills all properties of definition 10. We perform a case distinction on the rule used to type the reply.

- **(T-REDIR):** Property 1 follows directly from (T-REDIR) We perform a case distinction:
  - if \( t_b \neq \text{usr} \) Then we need to show property 2 which follows immediately from the typing rule, using lemma 15
  - if \( t_b = \text{usr} \) and \( \mu = \text{hon} \), then property 1 follows from (T-REDIR). Properties 5 and 7 are trivial. Property 8 follows from (T-REDIR).
  - if \( t_b = \text{usr} \) and \( \mu = \text{att} \) then properties 3 and 6 are trivial. Property 4 follows from (T-REDIR). Property 8 follows from (T-REDIR).

- **(T-REPLY):** Property 1 follows directly from (T-REPLY) and property 8 is trivial. We perform a case distinction:
  - if \( t_b \neq \text{usr} \) Then we need to show property 2 which follows immediately from the typing rule, using lemma 15
  - if \( t_b = \text{usr} \) and \( \mu = \text{hon} \), then properties 5, 7 and 4 of definition 10 follow from (T-REPLY) and (T-FORM)
  - if \( t_b = \text{usr} \) and \( \mu = \text{att} \) then properties 6 and 5 of definition 10 follow from (T-REPLY). Property 8 follows immediately from (T-REPLY) and from the observation that rule (T-BEREFFAIL) and (T-BASSIGNFAIL) are not used for typing the script, as the script can also be typed in the honest typing branch \( b = \text{hon} \).

- **(T-REPLY-EFF):** All claims are trivial.

We can now define the typing of websystems, which simply states that all browsers and servers contained in the system are well typed.
\( T_O = \{(t_b, n, u, l', \mu)\} \) for some \( l' \)

4) for all \( v^\tau \in K \) we have \( \tau \subseteq_{\ell a} \ell_a \)

Next, we show that any script created by the attacker, that is served over a low integrity network connection is well-typed in the users browser.

**Lemma 19** (Attacker Script Typability). For all scripts \( s \) and well formed environments \( \Gamma \), URLs \( u \) with \( I(\ell_a) \subseteq_I I(\lambda(u)) \), \( \forall n^\tau \in \text{values}(s), \tau \subseteq_{\ell a} \ell_a, \text{vars}(s) = \emptyset \) we have \( \Gamma, I(\ell_a), \text{csrf} \vdash_{\ell a, p, u} s \).

**Proof.** We first show that for all browser expressions \( be \) we have \( \Gamma, \text{csrf} \vdash_{\ell a} be : \ell_a \). We show the claim by induction over \( be \).

- \( be = r \).
  - if \( C(\lambda(r)) \vDash_C C(\lambda(u)) \) the claim follows immediately using rule (T-BEREFFAIL).
  - if \( C(\lambda(r)) \vDash_C C(\lambda(u)) \) we know by the well-formedness of \( \Gamma \) that \( C(\text{ref}_r(\Gamma R^o(r))) \vDash_C C(\ell_a) \) and hence also \( I(\ell_a) \subseteq_I I(\text{ref}_r(\Gamma R^o(r))) \) and the claim follows using (T-BEREF) and (T-BESUB).

- \( be = v^\tau \): The claim follows from the assumption \( \tau \subseteq_{\ell a} \ell_a \) and rule (T-BEVAL).

- \( be = \text{dom}(be') \): Immediately by rule (T-BEDOM).

- \( be = be_1 \circ be_2 \): By induction and rule (T-BEBINOP)

We now show the main claim by induction over \( s \).

\( s = s_1 ; s_2 \): the claim follows from the induction hypothesis for \( s_1 \) and \( s_2 \) and (T-BSEQ)

\( s = \text{skip} \): trivial with (T-BSKIP)

\( s = r := be \): We distinguish two cases

- if \( I(\lambda(u)) \nsubseteq_I I(\lambda(r)) \) then the claim is trivial with rule (T-BASSIGNFAIL).
- if \( I(\lambda(u)) \subseteq_I I(\lambda(r)) \) then we know because of \( I(\ell_a) \subseteq_I \lambda(u) \) that also \( I(\ell_a) \subseteq_I \lambda(r) \). Therefore, we know by well-formedness of \( \Gamma \) that if \( \Gamma R^o(r) = \tau \) with \( \tau = \text{cred}(\cdot) \) then \( C(\tau) \subseteq_C C(\ell_a) \). We can hence show \( \Gamma \vdash_{\ell a} r : \text{ref}(\ell_a) \).

Since we know that \( \Gamma, b \vdash_{\ell a} be : \ell_a \) the claim follows.

\( s = \text{set} \text{dom}(v, u, be') \): The claim follows from rule (T-BSETDOM), using our observation about expression types.

\( s = \text{include}(u, be) \): The claim follows from rule (T-BINCLUDE), using our observation about expression types.

Next, we show that the attacker can only learn low confidentiality values from the network.

**Lemma 20.** Attacker Knowledge for low confidentiality requests Whenever we have \( \alpha = \text{req}(\text{usr}, n, u, p, ck, o)^{\mu} \) with \( \Gamma \vdash_{\ell a, \text{usr}} \alpha \) and \( \lambda(u) \subseteq_C C(\ell_a) \) then for all \( n^\tau \in \text{ns}(p, ck) \) we have \( C(\tau) \subseteq_C C(\ell_a) \).

**Proof.** For \( \mu \) is hon the claim for \( \text{ns}(p) \) follows immediately from the well-formedness of URLs and property 1 of definition 9. Otherwise the claim follows directly from property 2 of definition 9.

The claim for \( \text{ns}(ck) \) follows immediately from property 3 of definition 9.

The next two lemmas show that requests and responses crafted by the attacker are well-typed.

**Lemma 21** (Attacker Request). Let \( \alpha = \text{req}(\text{usr}, n, u, p, ck, o)^{\perp, \att} \) with \( u_b \neq \text{usr} \) and for all \( v^\tau \in \text{values}(p, ck) \) we have \( \tau \subseteq_{\ell a} \ell_a \). Then \( \Gamma \vdash_{\ell a, \text{usr}} \alpha \).

**Proof.** Since \( u_b \neq \text{usr} \) and \( \mu \) is att, we have to show properties 2 and 3 of definition 9. Both claims follow immediately since \( \forall n^\tau \in \text{ns}(p, ck), \tau \subseteq_{\ell a} \ell_a \).

**Lemma 22** (Attacker Response). Let \( \alpha = \text{res}(\text{usr}, n, u, u', \nu, \sigma, \text{page}, s, p)^{\perp, \att} \) with \( I(\ell_a) \subseteq_I \lambda(u) \) and for all \( v^\tau \in \text{values}(p, ck) \) we have \( \tau \subseteq_{\ell a} \ell_a \). Then \( \Gamma \vdash_{\ell a, \text{usr}} \alpha \).

**Proof.** We show that \( \alpha \) fulfills the properties of definition 10.

We have to show properties 1, 6, 7, 8 and 9 of definition 10.

With \( \forall n^\tau \in \text{ns}(\nu, \sigma, \text{page}, s), \tau \subseteq_{\ell a} \ell_a \). Properties 7 and 8 are trivial. Property 9 follows immediately from lemma 19.

For property 4 we look at all \( r \in ck \) and perform a case distinction:

- If \( \lambda(u) \subseteq_I \lambda(r) \) then by transitivity of \( \subseteq_I \) we know \( I(\ell_a) \subseteq_I \lambda(r) \) and hence by well-formedness of \( \Gamma \) we know that if \( \Gamma R^o(r) = \text{cred}(\cdot) \) then \( C(\Gamma R^o(r)) \subseteq_C C(\ell_a) \). We hence know that \( \ell_a \subseteq_{\ell a} \text{ref}_r(\Gamma R^o(r)) \).

- If \( \lambda(u) \nsubseteq_I \lambda(r) \) then the property is trivially true.

For property 8 we have to show property 2 of definition 9 which follows immediately.

Finally, we show that whenever a well-formed system takes a step, it produces another well-typed system.
Lemma 23 (System Subject Reduction). Let $W$ be a websystem with $\Gamma \models_\ell,usr (\ell, K) \triangleright W$ and $(\ell, K) \triangleright W \xrightarrow{\alpha} (\ell', K') \triangleright W'$. Then we have $\Gamma \models_\ell,usr (\ell', K') \triangleright W'$.

Proof. We do a proof by a case analysis over the derivation of $\xrightarrow{\alpha}$.

- (A-NIL) If the step was taken using rule (A-NIL) then we perform an induction on the internal step. If the step is taken through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction. If it is taken locally in one browser or server the claim follows from lemma 6 or lemma 14 and the fact that $K = K'$. Property 3 of definition 13 follows from the observation that raising the server integrity label can only happen in rule (S-RESET).

- (A-BROSER) Follows immediately from lemma 7 and lemma 17 Property 3 follows from the semantics rules for browsers and servers. Property 4 follows from lemma 20.

- (A-SERBRO) Follows immediately from lemma 13 and lemma 8 We can apply lemma 8 because of property 3 of definition 13.

- (A-TIMEOUTSEND) Follows immediately from lemma 7.

- (A-TIMEOUTREC) Let $\alpha'$ be the response sent in the rule. Then we trivially have $\Gamma \models_\ell,usr \alpha'$ and the claim follows using lemma 8 and property item 3 of definition 13.

- (A-BROATK) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \text{res}((n, u, p, ck, a)^{l,\mu}$ and $I(\ell_a) \subseteq I(\lambda(u))$. The typing of the browser follows immediately from lemma 7 Property 4 follows from lemma 20.

- (A-ATKSER) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \text{req}(t_b, n, u, p, ck, o)^{l,\mu}$, where $ns(p, ck) \subset K$. We hence get by lemma 21 that $\Gamma \models_\ell,usr \alpha$ and the claim follows from lemma 17.

- (A-SERATK) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \text{res}((t_b, n, u, u', \nu, \mu, page, s)^{l,\mu}$, where $n \in K$. From lemma 13 we get that the resulting server state is well-typed and that $\alpha$ is a well-typed response. We now have to show that for all $\tau \in \tau'$ we have $\tau \subseteq_\ell \ell_a$. Since $n \in K$ and the only point where the attacker can learn $n$ is in rule (A-ATKSER) we know that $\ell_b \neq \ell$ and $\mu = \ell$. The claim follows directly from property 2 of definition 10.

- (A-ATK BRO) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \text{res}((t_b, n, u, u', \nu, \mu, page, s)^{l,\mu}$ where $t_b = \ell$, $l = l$, $\mu = \ell$ and $I(\ell_a) \subseteq I(\lambda(u))$. By lemma 22 we get that $\alpha$ is a well-typed response, then the claim follows from lemma 8 (which we can apply, since $\mu = \ell$ and $l = l$).

$\square$

K. Relation

We now define a notion of High Equality between different components, that will be used to relate two websystems.

The general intuition is, that everything that is of high integrity must be equal, while values of low integrity can be arbitrarily different.

Definition 14 (High Equality). We define high equality in different contexts:

1) For two (browser or server) expressions $e, e'$ we inductively define $e =_{\perp_1} e'$ by the following rules.

\[
\begin{align*}
e =_{\perp_1} e & \quad \frac{I(\ell_a) \subseteq I(\tau) \cap I(\tau')}{v^\tau =_{\perp_1} v'^{\tau'}} & \quad \frac{\text{fresh}(\tau) =_{\perp_1} \text{fresh}(\tau')}{\ell =_{\perp_1} \ell'} & \quad \frac{e_1 =_{\perp_1} e_2}{e_1 \cap e_2 =_{\perp_1} e_2 \cap e_2'} & \quad \frac{e_2 =_{\perp_1} e_2'}{e_2 \cap e_2 =_{\perp_1} e_2 \cap e_2'} \\
e_1 =_{\perp_1} e_1' & \quad \frac{\text{dom}(e_1, e_2) =_{\perp_1} \text{dom}(e_1', e_2')}{e =_{\perp_1} e'} & \quad \frac{|\vec{e}| = |\vec{e}'|}{\forall i \in [1, \ldots, |\vec{e}|]. e_i =_{\perp_1} e_i'} & \quad \frac{e_i =_{\perp_1} e_i'}{\vec{e} =_{\perp_1} \vec{e}'}
\end{align*}
\]

2) For two pages, $\text{page}, \text{page}'$ we define $\text{page} =_{\perp_1} \text{page}'$ as

\[
\forall v \in \text{dom}(\text{page}). \text{page}(v) = \text{form}(u, v^\ell, \lambda, \text{page}'(v)) = \text{form}(u', v'^\ell, \lambda, \text{page}'(v)) \wedge u =_{\perp_1} u' \wedge v_i =_{\perp_1} v_i'
\]

\[
\text{page} =_{\perp_1} \text{page}'
\]

3) For two scripts, $s, s'$ we define $s =_{\perp_1} s'$ as

\[
\text{skip} =_{\perp_1} \text{skip} & \quad \frac{s_1 =_{\perp_1} s_2}{s_1; s_2 =_{\perp_1} s_2'} & \quad \frac{s_2 =_{\perp_1} s_2'}{s_1; s_2 =_{\perp_1} s_2' ; s_2'} & \quad \frac{\text{be} =_{\perp_1} \text{be}'}{\text{be} =_{\perp_1} \text{be}'} & \quad \frac{\text{be} =_{\perp_1} \text{be}'}{\text{be} =_{\perp_1} \text{be}'}
\]

\[
\text{include}(u, \text{be}) =_{\perp_1} \text{include}(u, \text{be}') & \quad \frac{\text{be} =_{\perp_1} \text{be}'}{\text{be} =_{\perp_1} \text{be}'} & \quad \frac{\text{be} =_{\perp_1} \text{be}'}{\text{be} =_{\perp_1} \text{be}'}
\]

\[
\text{setdom}(\text{be}, u, \text{be}) =_{\perp_1} \text{setdom}(\text{be}', u, \text{be}')
\]

4) For two commands \( c, c' \) we define \( c =_{\bot} c' \) as

\[
\begin{array}{cccc}
\text{skip} =_{\bot} \text{skip} & \text{halt} =_{\bot} \text{halt} & c_1 =_{\bot} c_2 & c_2 =_{\bot} c'_2 \\
\text{if } s.e \text{ then } c_1 \text{ else } c_2 =_{\bot} \text{if } s.e' \text{ then } c'_1 \text{ else } c'_2 \\
se =_{\bot} s.e' & \forall i \in [1 \ldots |s.e|], s.e_i =_{\bot} s.e'_i \\
\text{start } s.e =_{\bot} \text{start } s.e' & \text{auth } s.e \text{ at } l =_{\bot} \text{auth } s.e' \text{ at } l \\
\text{reply } (p, c, k) \text{ with } f = s.e =_{\bot} \text{reply } (p, s, c) \text{ with } f = s.e' \\
\text{redirect } (u, z, c) \text{ with } f = s.e =_{\bot} \text{redirect } (u, z, c) \text{ with } f = s.e' \\
\end{array}
\]

5) For two memories \( M, M' \) we write \( M =_{\Gamma, \bot, M'} \), if

- for all \( r \in \text{dom}(M) \cup \text{dom}(M') \) we have \( M(r) =_{\bot} M'(r) \)
- for all \( r \in \text{dom}(M) \setminus \text{dom}(M') \) with \( M(r) = v^r \) we have \( I(\ell_a) \subsetneq l \tau \)
- for all \( r \in \text{dom}(M') \setminus \text{dom}(M) \) with \( M'(r) = v^r \) we have \( I(\ell_a) \subseteq l \tau \)

6) For two requests \( \alpha = \text{req}(t_b, n, u, u, p, c, k, o)^{1,\mu} \) (resp. \( \alpha = \text{req}(t_b, n, u, p, c, k, o)^{1,\mu} \)) and \( \beta = \text{req}(t_b', n', u', p', c', o')^{1,\mu'} \) (resp. \( \beta = \text{req}(t_b', n', u', p', c', o')^{1,\mu'} \)) we let

\[
\alpha =_{\bot} \beta \iff \text{I(\ell_a) \nsubseteq l} l \subseteq l' \Rightarrow \begin{align*}
t_b &= t_b' \land n = n' \land u = u' \\
\text{dom}(p) &= \text{dom}(p') \land \forall x \in \text{dom}(p), p(x) =_{\bot} p'(x) \\
ck &=_{\Gamma, \bot} c' \\
\land l &= l' \land \mu = \mu'
\end{align*}
\]

7) For two responses \( \alpha = \text{res}(t_b, n, u, u, v, c, k, p, a, s)^{1,\mu} \) (resp. \( \alpha = \text{res}(t_b, n, u, u, v, c, k, p, a, s)^{1,\mu} \)) and \( \beta = \text{res}(t_b', n', u', u', v', c', k', p, a', s')^{1,\mu'} \) (resp. \( \beta = \text{res}(t_b', n', u', u', v', c', k', p, a', s')^{1,\mu'} \)) we let

\[
\alpha =_{\bot} \beta \iff \begin{align*}
ck &=_{\Gamma, \bot} c' \\
\text{I(\ell_a) \nsubseteq l} l \subseteq l' & \Rightarrow \begin{align*}
t_b &= t_b' \land n = n' \land u = u' \land u = u' \land v =_{\bot} v' \\
page &=_{\bot} page' \land s = s' \land \\
l &= l' \land \mu = \mu'
\end{align*}
\end{align*}
\]

8) For two authentication events \( \alpha = \text{auth}(t_b, a, c) \) and \( \alpha' = \text{auth}(t_b', a', c') \) we let \( \alpha =_{\bot} \alpha' \) if

- \( I(\ell_a) \subseteq l \) and \( I(\ell_a) \subseteq l' \) or
- \( \alpha = \alpha' \)

We introduce a predicate \( \text{bad}(\cdot) \) which we use to denote that the system has entered a state in which the browser will perform no more actions because it received an error message from the server.

**Definition 15 (Bad State).** A browser \( B = (N, K, P, T, Q, o)^{\text{usr}, \mu} \) is in a bad state and we write \( \text{bad}(B) \) if \( \text{halt} \in \delta \).
- A server \( S = (D, \phi, t) \) is in a bad state and we write \( \text{bad}(S) \) if there is a \( t \in \text{running}(S) \) with \( t = [c]^{1,\mu}_{E,R} \) and
  - \( t_b = \text{usr} \)
  - \( c = \text{reply} (\text{error}, \text{skip}, \{\}) \) or \( c = \text{bad} \)
- A web system \( (\ell_a, K) \triangleright W \) is in a bad state and we write \( \text{bad}(A) \) if
  - with \( \{B\} = \text{browsers}(W) \), we have \( \text{bad}(B) \)
- for any $S \in \text{servers}(W)$, we have $\text{bad}(S)$

The following properties are straightforward by inspecting the semantic rules:

- If $\text{bad}(A)$ and $A \xrightarrow{\circ} A'$, then $\text{bad}(A')$.
- If $\text{bad}(A) \land A \xrightarrow{\circ} A'$, then there does not exist $\xi[\varphi_{\ell_1 \ell_2 \ell_3}] \in \alpha$ with $I(\ell_a) \not\subseteq I(\ell)$.

We now define a relation between two browsers.

**Definition 16 (Browser Relation).** Let $B = (N, K, P, T, Q, \alpha)_{\varphi l, i, \mu}$ and $B' = (N', K', P', T', Q', \alpha')_{\varphi l, i', \mu'}$ be browsers. Then we write $B \approx B'$ if the following conditions hold:

1) $\Gamma \vdash_{\alpha, \varphi l} B$ and $\Gamma \vdash_{\alpha, \varphi l} B'$
2) $I(\ell_a) \subseteq I$ $\iff$ $I(\ell_a) \subseteq I'$ and $I(\ell_a) \not\subseteq I \Rightarrow l = l'$
3) If $I(\ell_a) \not\subseteq I$ then $N = N'$
4) $\Gamma =_{\alpha l, \varphi l, \mu} K'$
5) For all $t \in dom(P)$ if $P(t) = (u_1, page1, l_1, \mu_1)$ then $I(\ell_a) \subseteq I$, $l_1$ or $t \in dom(P')$ with $P'(t) = (u_2, page2, l_2, \mu_2)$ and $u_1 = u_2$, page1 = page2 and vice versa.
6) If $I(\ell_a) \not\subseteq I$ then $dom(T) = dom(T')$ and if $T = \{t \mapsto s\}$ and $T' = \{t \mapsto s'\}$ then $s =_{\ell I} s'$
7) $\overline{a} = a'$
8) If $I(\ell_a) \not\subseteq I$ and $Q = \{\alpha\}$ then $Q' = \{\alpha'\}$ with $\alpha =_{\ell I} \alpha'$.

We let $B \approx B'$ if

- $\text{bad}(B)$
- or $B \approx B'$

Intuitively, two browsers are related by the relation $\approx B'$ if

1) Both browsers are well typed.
2) Either both have low or high integrity. If the integrity is high, it must be the same.
3) If the integrity is high, then the network connections are equal.
4) The cookie jars fulfill high equality.
5) For any high integrity page in a tab of one browser, there exists a page in the same tab of the other browser, with same URL, integrity and attacked mode, and a DOM that fulfills high equality.
6) For high integrity browsers the scripts fulfill high equality.
7) The list of user actions is equal.
8) If the browsers are in high integrity states, then the events in the output buffer must fulfill high equality.

We then define the relation $\approx B_1$, which holds if the left browser is in a bad state, or the browsers are in the relation $\approx B'$.

We then show that the relation $\approx B_1$ is symmetric and transitive. Note that this does not hold for $\approx B_2$.

**Lemma 24 ($\approx B_1$ is symmetric and transitive).** The relation $\approx B_1$ is symmetric and transitive.

**Proof.** Trivial, by checking the individual properties of definition 16.

Next, we show that high equality on browser expressions is preserved under evaluation in the browser.

**Lemma 25 (Preservation of $=_{\perp I}$ under browser evaluation).** Let $be$ and $be'$ be browser expressions with $be =_{\perp I} be'$, let $M, M'$ be memories with $M =_{\alpha l, \varphi l, \mu} M'$, let $u$ be a URL and let $page = f$ and $page' = f'$ be pages with $page =_{\perp I} page'$. Then $\text{eval}_{\lambda(u)}(be, M, f) =_{\perp I} \text{eval}_{\lambda(u)}(be', M', f')$.

**Proof.** Let $\lambda v' = \text{eval}_{\lambda(u)}(be, M, f)$ and $\lambda v' = \text{eval}_{\lambda(u)}(be', M', f')$. If $I(\ell_a) \not\subseteq I$ then $I(\ell_b) \not\subseteq I$ the claim is trivial. We hence now assume $I(\ell_a) \subseteq I(\ell_b) \cap I(\ell_b)$, i.e., $I(\ell_a) \not\subseteq I(\ell_b) \lor I(\ell_b) \not\subseteq I(\ell_b)$.

- $be = x$ : Impossible, since evaluation is not defined on variables.
- $be = v^\circ$ : Trivial, since evaluation on values is the identity (BE-VAL).
- $be = be_1 \circ be_2$: Then $be' = be_1' \circ be_2'$ with $be_3 =_{\perp I} be_1'$ and $be_2 =_{\perp I} be_2'$. Let $v_1^\circ = \text{eval}_{\lambda(u)}(be_1, M, f)$, let $v_2^\circ = \text{eval}_{\lambda(u)}(be_2, M, f)$, let $v_1^\circ = \text{eval}_{\lambda(u)}(be_1', M', f')$ and let $v_2^\circ = \text{eval}_{\lambda(u)}(be_2', M', f')$. By induction we get $v_1^\circ =_{\perp I} v_1^\circ$ and $v_2^\circ =_{\perp I} v_2^\circ$. By rule (BE-BINOP) we know that $I(\tau) = I(\tau_1) \cap I(\tau_2)$ and $I(\tau') = I(\tau_1') \cap I(\tau_2')$. We know that $I(\ell_a) \not\subseteq I(\ell_b) \lor I(\ell_a) \not\subseteq I(\ell_b)$.

- If $I(\ell_a) \not\subseteq I(\ell_b)$ then we know that $I(\ell_a) \subseteq I(\tau_1)$ and $I(\ell_a) \not\subseteq I(\tau_2)$. By the definition of $=_{\perp I}$ we then know that $v_1^\circ = v_1^\circ$ and $v_2^\circ = v_2^\circ$ and we get that $v^\circ = v'^\circ$.

- If $I(\ell_a) \not\subseteq I(\ell_b)$ the claim follows analog.
• be = dom(be₁, be₂): Then be' = dom(be₁', be₂') with be₁ = ⊥_I be₁' and be₂ = ⊥_I be₂'. Let v₁ᵀ₁ = evalₜ(u)(be₁, M, f), let v₂ᵀ₂ = evalₜ(u)(be₂, M, f), let v₁ᵀ₁' = evalₜ(u)(be₁', M', f') and let v₂ᵀ₂' = evalₜ(u)(be₂', M', f'). By induction we get v₁ᵀ₁ = ⊥_I v₁ᵀ₁' and v₂ᵀ₂ = ⊥_I v₂ᵀ₂'.

We distinguish the following cases:
- If I(ℓₐ) ⊆ₜ I(τ₁) ∪ₜ I(τ₂) then by the definition of =ₜ we also know that I(ℓₐ) ⊆ₜ I(τ₁) ∪ₜ I(τ₂'). Then the claim is trivial, since then I(ℓₐ) ⊆ₜ I(τ₁) ∪ₜ I(τ₂') by (BE-Dom).
- If I(ℓₐ) ⊆ₜ I(τ₂) ∪ₜ I(τ₂') the claim follows analogous to the previous one.
- If I(ℓₐ) ⊆ₜ I(τ₁), I(ℓₐ) ⊆ₜ I(τ₁'), I(ℓₐ) ⊆ₜ I(τ₂) and I(ℓₐ) ⊆ₜ I(τ₂') Then we know by that v₁ = v₁' and v₂ = v₂'. The claim then follows from page =ₜ page' and rule (BE-Dom).

Now we introduce the notion of deterministic termination. This property states that a system terminates and can only produce a single trace. This is a property that holds in the honest run, as the assumptions on user behaviour allow only terminating runs and without the attacker there is no point of non-determinism.

**Definition 17 (Deterministic Termination).** We say that a websystem W is deterministically terminating for a userusr if there exists exactly one unattacked trace γ such that (ℓₐ, K) ⊢ W ̸→ γ (ℓₐ, K') ⊢ W' where W = Busr(M', P', {})) || W' and browsers(W') = ∅ for some K', W', M', P'.

We say that a server thread t = [c]ₑᵣ is deterministically terminating if there exists exactly one ℧ with t ⊢ ℧ for some ℧ ∈ [c]ₑᵣ; with
- c' = reply (page, ·, ·) with x = ·, where page ≠ error
- or c' = redirect (·, ·, ·) with x = ·

We say that server S is deterministically terminating if all t ∈ running(S) are deterministically terminating.

Note that it immediately follows that all servers in a deterministically terminating web system are also deterministically terminating.

Next we define a relation between two servers:

**Definition 18 (Server Relation).** Let S = (D, ϕ, t) and S' = (D', ϕ', t') Then we write S ≃ₘ S' if or the following conditions hold

1. Γ ⊨ₜ usr S and Γ ⊨ₜ usr S'
2. Let t₁ᵀ₁ = {t₁|t₁ ∈ running(t) ∧ I(ℓₐ) ⊆ₜ intᵣ₁(t₁)} and t₂ᵀ₂ = {t₂|t₂ ∈ running(t) ∧ I(ℓₐ) ⊆ₜ intᵣ₂(t₂)}. Then there is a bijection c : t₁ᵀ₁ → t₂ᵀ₂ such that for all t₁ ∈ t₁ᵀ₁ and t₂ = c(t₁) ∈ t₂ᵀ₂, if we let t₁ = [c₁]ₑ₁ᵣ₁ and t₂ = [c₂]ₑ₂ᵣ₂, then we have
   a) R₁ = R₁ and with E₁ = i₁, j₁ and E₂ = i₂, j₂ we have j₁ = ⊥₁, j₂.
   b) With E₁ = (i₁, j₁) we have D₀₁(i₁) =Γ₁,₁ D₀₂(i₁)
   c) the following holds:
      i) I(ℓₐ) ⊆ₜ l₁ ⇒ I(ℓₐ) ⊆ₜ l₂ and I(ℓₐ) ⊆ₜ l₁ ⇒ l₁ = l₂
      ii) if I(ℓₐ) ⊆ₜ l₁ then c₁ = ⊥₁, c₂
      iii) if I(ℓₐ) ⊆ₜ l₁ and there exists an l with I(ℓₐ) ⊆ₜ l and c₁' and c₂' such that c₁ = c₁'; reset l; c₁' then there exist c₂' and c₂'' such that c₂' = c₂'' and vice versa.
3. We have
   a) for all j₁ ∈ dom(D₁) ∩ dom(D₂) with C(τ) ⊆ₜ C(τ) that Dₘ(j₁) =Γ₁,₂ Dₘ(j₁)
   b) for all j₁ ∈ dom(D₁) ∩ dom(D₂) with C(τ) ⊆ₜ C(τ) and for all r ∈ R with I(ℓₐ) ⊆ₜ Γᵣₘ(r) we have Dₘ(j₁)(r) = ⊥₁
   c) for all j₁ ∈ dom(D₁) ∩ dom(D₂) with C(τ) ⊆ₜ C(τ) and for all r ∈ R with I(ℓₐ) ⊆ₜ Γᵣₘ(r) we have Dₘ(j₁)(r) = ⊥₁
4. For all j₁ with I(ℓₐ) ⊆ₜ τ we have that φ(j₁) = φ'(j₁).

We let S ≃ₘ S' if

- bad(S)
- or S ≃ₘ S' and S' is deterministically terminating.

Intuitively, two servers are in the relation ≃ₘ if
1) Both servers are well-typed.
2) There is a bijection between high integrity running threads on the two servers. For each pair we have that
   a) They have the same request context and global memory index. For high integrity threads they also have the same session memory index.
b) High equality holds between the two global memories.

c) i) Either both threads have high or both have low integrity. If it is high it has to be equal.

ii) For high integrity threads, the two codes have to be high equal

iii) If the integrity of one thread is low, but it can be raised to high using a reset command, then there also has to be a reset with the same high integrity label in the other thread.

3) • For all session identifiers appearing in both threads that are secret, the session memories indexed by the identifiers are high equal
   • For all session identifiers present in only one thread, that are secret, all high integrity references are unset.

4) For all high integrity session identifiers, the user information (ϕ) is equal.

We then define the relation ≃I which holds if the left server is in a bad state, or the servers are in the relation ≃S and the right server is deterministically terminating.

We then show that the relation ≃I is symmetric and transitive. Note that this does not hold for ≃S.

Lemma 26 (≃S is symmetric and transitive). The relation ≃S is symmetric and transitive.

Proof. Trivial, by checking the individual properties of definition 16.

Next, we show that high equality for server expressions is preserved under evaluation.

Lemma 27 (Preservation of =⊥ under server evaluation). Let se and se′ be server expressions with se =⊥ se′, let E = i, j and E′ = i′, j′ with j =⊥ j′, let D, D′ be databases and ΓR8, ΓR8 typing environments with Dθ(i) =ΓR8,⊥ Dθ(i′) and if j ≠ ⊥ then Dθ(j) =ΓR8,⊥ Dθ(j′) with ∀r. ΓR8(r) = ΓR8(r)⊔label(j). Then evalE(se, D) =⊥ evalE(se′, D′).

Proof. Let vτ = evalE|se, D) and vτ′ = evalE|se′, D′). If I(ℓa) ≺I I(τ) ≺I I(τ′) the claim is trivial. We hence now assume I(ℓa) ⊑I I(τ) ⊑I I(τ′), i.e., I(ℓa) ⊑I I(τ) ⊓I I(τ′)

• be = x : Impossible, since evaluation is not defined on variables.

• be = vτ′ : Trivial, since evaluation on values is the identity (E-VAL)

• be = fresh(τ) : Then be′ = fresh(τ′). By rule (SE-FRESH) we know that v, v′ ∈ Nτ. For simplicity, we assume that v = v′ and that the sampled names are fresh (i.e., have not been sampled before and will not be sampled again).

• se = se1 ⊓ se2: Then se′ = se′ 1 ⊓ se′ 2 with se1 =⊥ se′ 1 and se2 =⊥ se′ 2 and the claim follows by induction analog to lemma 25.

• se = @r: Then se′ = @r. The claim then follows from Dθ(i) =ΓR8,⊥ Dθ(i′).

• se = $\tau$: Then se′ = $\tau$ and the claim then follows from Dθ(j) =ΓR8,⊥ Dθ(j′).

Now we introduce a relation between websystems:

Definition 19 (Integrity Relation). Given a typing environment Γ, we consider two websystems A = (ℓa, K) ▷ W and A′ = (ℓa′, K′) ▷ W′ to be in the integrity relation ≃I if

1) I ⊢ℓa, ur Γ and I ⊢ℓa′, ur Γ

2) For each server S ∈ servers(W) there exists exactly one server S′ ∈ servers(W′) such that urls(S) = urls(S′) and vice versa, i.e. the available URLs and the code associated to them are the same in both web systems. We will call these servers S and S′ corresponding servers. Formally, the correspondence is a bijection between the sets servers(W) and servers(W′).

3) For all servers S in W and the corresponding servers S′ in W′ we have that S ≃S S′

4) W contains exactly one browser B = (N, K, P, T, Q, a)ur, 1,t, u′, and W′ contains exactly one browser B′ = (N′, K′, P′, T′, Q′, a′)ur, 1,t′, u′, and we have B ≃B B′.

We furthermore let A ≃I A′ if

1) bad(A)

2) or A ≃I A′ and A′ is deterministically terminating.

Intuitively, we require that

1) Both websystems are well-typed

2) All servers have a matching server in the other websystem that contains the same URLs and commands (i.e., statically the websystems are equal)

3) All corresponding servers are in the relation ≃S.

4) Both websystems contain exactly one browser, and they are in the relation ≃B.

We then define the relation ≃I, which holds if the left system is in a bad state, or the systems are in the relation ≃I and the right system is deterministically terminating.
We show that all the properties of definition 16 are fulfilled. In all cases property 1 follows immediately from lemma 6.

**Proof.** Trivial, by inspecting the single conditions.

Now we show that whenever a browser processes an event with low sync integrity for an internal step, then the state before and after taking the step are in the relation.

**Lemma 29 (Low Sync Integrity Browser Steps).** Let \( B = (N, K, P, T, Q, \vec{a}) \) and \( B' = (N', K', P', T', Q', \vec{a}') \) be browsers with \( B \triangleright \tau \mapsto \tau' \) and \( I(\ell_a) \subseteq I l' \) and \( \Gamma \models_{\ell_a, \text{usr}} B \). Then \( B \triangleright B' \).

**Proof.** We show that all the properties of definition 16 are fulfilled. In all cases property 1 follows immediately from lemma 6.

Proof by induction over the derivation of the step \( \alpha \)

- (B-SEQ) follows from induction.
- (B-Skip) Properties 2, 3, 4, 5, 7 and 8 are trivial, since \( l = l' \), \( P = P' \), \( K = K' \), \( N = N' \), \( \vec{a} = \vec{a}' \), and \( Q = Q' \). Property 6 is trivial, since because of \( I(\ell_a) \subseteq I \text{sync}_l(\alpha) \) we know \( I(\ell_a) \subseteq I l \).
- (B-END) Impossible, since \( I(\ell_a) \subseteq I \text{sync}_l(\alpha) \).
- (B-SetReference) Properties 2, 4, 3, 7 and 8 are trivial, since \( l = l' \), \( P = P' \), \( K = K' \), \( N = N' \), \( \vec{a} = \vec{a}' \), and \( Q = Q' \). Property 6 is trivial, since because \( I(\ell_a) \subseteq I \text{sync}_l(\alpha) \) we know \( I(\ell_a) \subseteq I l \).
- (B-Load): Impossible since \( I(\ell_a) \subseteq I l \)
- (B-Submit): Impossible since \( I(\ell_a) \subseteq I l \)
- (B-Include): Properties 2, 4, 3 and 7 are trivial, since \( l = l' \), \( P = P' \), \( K = K' \), \( N = N' \), \( \vec{a} = \vec{a}' \). Properties 6 and 8 are trivial, since because \( I(\ell_a) \subseteq I \text{sync}_l(\alpha) \) we know \( I(\ell_a) \subseteq I l \).

We show the same for internal steps on the server side with low sync integrity.

**Lemma 30 (Low Sync Integrity Server Steps).** Let \( S = (\ell, \phi, t) \) and \( S = (\ell', \phi', t') \) with \( S \overset{\alpha}{\rightarrow} S' \) and \( \alpha \in \{\bullet, \text{error}\} \) and \( I(\ell_a) \subseteq I \text{sync}_l(\alpha) \) and \( \Gamma \models_{\ell_a, \text{usr}} S \). Then \( S \overset{\alpha}{\rightarrow} S' \).

**Proof.** We show that all the properties of definition 18 are fulfilled. In all cases property 1 follows immediately from lemma 14.

Let \( t_1 \in \text{running}(S) \) and \( t_1 = [c]_{E,R}^{\ell_a, \mu} \). Then there exists \( t_1' \in \text{running}(S') \) with \( t_1' = [c']_{E,R}^{\ell_a, \mu} \).

We prove the claim by induction over the derivation of step \( \alpha \).

- (S-SEQ) The claim follows from the induction hypothesis.
- (S-IfTrue) Properties 2a, 2b, 3 and 4 are trivial since \( I(\ell_a) \subseteq I l \), \( E = E' \), \( D = D' \), \( \phi = \phi \). Property 2c is trivial since \( I(\ell_a) \subseteq I l \) and \( I(\ell_a) \subseteq I l' \).
- (S-TcterTrue) All properties are trivial since \( I(\ell_a) \subseteq I l \), \( E = E' \), \( D = D' \), \( l = l' \) and \( \phi = \phi \).
- (S-Skip) All properties are trivial since \( I(\ell_a) \subseteq I l \), \( E = E' \), \( D = D' \), \( l = l' \) and \( \phi = \phi \).
- (S-Reset) Then \( c = \text{reset} l'' \) Properties 2a, 2b, 3 and 4 are trivial since \( E = E' \), \( D = D' \), \( l = l' \) and \( \phi = \phi \). Property 2c is trivial since \( I(\ell_a) \subseteq I l \) (because the reset command will never lower the integrity label) and \( I(\ell_a) \subseteq I l' \).
- (S-IfFalse) Analog to rule (S-IfTrue).
- (S-TcFalse) All properties are trivial since \( I(\ell_a) \subseteq I l \), \( E = E' \), \( D = D' \), \( l = l' \) and \( \phi = \phi \). Property (iii) of 2c does hold since we know that a token check is never followed by a reset – this is enforced in (S-IfFalse) and (S-IfTrue).
- (S-RestoreSession) Properties 2b, 2c, 3 and 4 are trivial since \( D = D' \), \( l = l' \) and \( \phi = \phi \). Let \( E = i, j \) and \( E' = i, j' \). If for \( t_1 \) we do not have \( I(\ell_a) \subseteq I \text{intr}_{i}(t_1) \), property 2a is trivial. Otherwise, we know that \( \mu = \text{hon} \). Then we know by rule (T-Running) and (T-Start) that because of \( I(\ell_a) \subseteq I l \) we have \( I(\ell_a) \subseteq I \text{js}_{i}(j) \) and \( I(\ell_a) \subseteq I \text{js}_{j}(j') \).

Property 2a immediately follows.
• (S-NEWSESSION) Properties 2b, 2c and 4 are trivial since with $D = (D_\ell, D_\phi)$, $D' = (D'_\ell, D'_\phi)$ we have $D_\ell = D'_\ell$, $l = l'$ and $\phi = \phi$. If for $t_1$ we do not have $I(\ell_{a_1}) \not\subseteq I(\ell_{a_1})$, property 2a is trivial. Otherwise, we know that $\mu = \text{hon}$. Then we know by rule (T-RUNNING) and (T-START) that because of $I(\ell_{a_1}) \subseteq I$ we have $I(\ell_{a_1}) \subseteq I(\text{label}(j))$ and $I(\ell_{a_1}) \subseteq I(\text{label}(j'))$. Property 2a immediately follows.

Let $E = i, j$. For property 3 we know that $j' \in D'_\phi \setminus D_\phi$. Since for all $r$, $(D'_\phi(j')(r)) = \perp (D'_\phi(j')$ is a fresh memory) property 3 immediately follows.

• (S-SETGLOBAL) Then we have $c = @r := se$. Properties 2a, 2b, 2c and 4 are trivial since with $D = (D_\ell, D_\phi)$, $D' = (D'_\ell, D'_\phi)$ we have $E = E'$, $D_\phi = D'_\phi$, $l = l'$ and $\phi = \phi$.

Let $E = i, j$ and let $v^r = D_\phi(i)(r)$ and $v'^{r'} = D'_\phi(i)(r)$. By rule (T-SETGLOBAL) we know that $I(\ell_{a_1}) \subseteq I(\text{ref}_r(\Gamma_{\text{R}_a}(r)))$. By property 1 we know that $\tau = \text{ref}_r(\Gamma_{\text{R}_a}(r)) \approx \tau'$. We hence have $I(\ell_{a_1}) \subseteq I(\tau)$ and $I(\ell_{a_1}) \subseteq I(\tau')$ and the claim follows immediately.

• (S-SETSESSION) Then we have $c = $ se. Properties 2a, 2b, 2c and 3 are trivial since $E = E'$, $D = D'$ and $l = l'$.

By rule (T-SETSESSION) with $I_{\ell_{a_1}} = \phi$, $\phi$ we get that $I(\ell_{a_1}) \subseteq I(\tau)$. Property 4 then follows immediately using lemma 5.

• (S-AUTH) Properties 2a, 2b, 2c and 3 are trivial since $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

• (S-OCHKSUCC) All properties are trivial since $I(\ell_{a_1}) \subseteq I(\tau)$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

• (S-OCHKFAIL) All properties are trivial since $I(\ell_{a_1}) \subseteq I(\tau)$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

• (S-LPARALLEL) The claim follows from induction hypothesis

• (S-RPARALLEL) The claim follows from induction hypothesis

Now, we show the same for browsers issuing a request with low sync integrity.

Lemma 31 (Low Sync Integrity Browser Request). Let $B = (N, K, P, T, Q, a^{\text{usr}}, l, \mu)$ and $B' = (N', K', P', T', Q', a'^{\text{usr}}, l', \mu')$ be browsers with $B \overset{\alpha}{\rightarrow} B'$ and $I(\ell_{a_1}) \subseteq I(\text{sync}_1(\alpha))$ and $\Gamma \models_{\ell_{a_1}} B$ and $\alpha = \text{res}(l_b, n, u, p, a, \text{ck})^{l''}$. Then $B \approx B'$ and $\text{sync}_1(\alpha) \subseteq I(l')$.

Proof. We show that all the properties of definition 16 are fulfilled. We know that $\alpha$ has been produces using rule (B-FLUSH). Property 1 follows immediately from lemma 7.

Properties 2, 3, 4, 5 and 7 are trivial, since $l = l'$, $P = P'$, $K = K'$, $N = N'$, $T = T'$ and $a = a'$. Properties 8 is trivial, since because $I(\ell_{a_1}) \subseteq I(\text{sync}_1(\alpha))$ we know $I(\ell_{a_1}) \subseteq I(l')$.

The claim $\text{sync}_1(\alpha) \subseteq I(l')$ follows immediately, by inspecting the rules (B-LOAD), (B-INCLUDE), (B-SUBMIT) and (B-REDIRECT).

Next, we show the same for a browser receiving a response with low sync integrity.

Lemma 32 (Low Sync Integrity Browser Response). Let $B = (N, K, P, T, Q, a^{\text{usr}}, l, \mu)$ and $B' = (N', K', P', T', Q', a'^{\text{usr}}, l', \mu')$ be browsers with $B \overset{\alpha}{\rightarrow} B'$ and $I(\ell_{a_1}) \subseteq I(\text{sync}_1(\alpha))$ and $\alpha = \text{res}(l_b, n, u, u', v, \text{ck}, p, a, s)^{l''}$ and $\Gamma \models_{\ell_{a_1}} B$ and $\Gamma \models_{\ell_{a_1}} \alpha$. Then $B \approx B'$ and $\text{sync}_1(\alpha) \subseteq I(l'')$.

Proof. We show that all the properties of definition 16 are fulfilled. We perform a case distinction on the rule used to derive $\alpha$.

In all cases for property 4 we get from property 3 of lemma 10 that for all updated references $r$, we have $I(\ell_{a_1}) \subseteq I(\text{ref}_r(\Gamma_{\text{R}_a}(r)))$. The claim then follows from property 2 for $B$ and $B'$.

• (B-RECVLOAD): The claim $\text{sync}_1(\alpha) \subseteq I(l'')$ follows from the observation that the integrity can only be lowered between the request and the response.

Property 1 follows immediately from lemma 8.

Property 7 is trivial, since we have $a = a'$.

Properties 2, 3, 5 and 8 are trivial, since because $I(\ell_{a_1}) \subseteq I(\text{sync}_1(\alpha))$ we know $I(\ell_{a_1}) \subseteq I(l)$ and $I(\ell_{a_1}) \subseteq I(l')$.

Property 9 follows immediately from $I(\ell_{a_1}) \subseteq I(l'')$, which we get from $\text{sync}_1(\alpha) \subseteq I(l'')$.

• (B-RECVINCLUDE) The claim $\text{sync}_1(\alpha) \subseteq I(l'')$ is trivial.
Property 1 follows immediately from lemma 8.
Properties 5 and 7 are trivial, since we have \( P = P' \) and \( \tilde{a} = \tilde{a'} \).
Properties 3, 6, 8 are trivial, since because \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \).

(B-REDIR) The claim \( sync_I(\alpha) \subseteq I \) \( I' \) is trivial.
Property 1 follows immediately from lemma 8.
Properties 5 and 7 are trivial, since we have \( P = P' \) and \( \tilde{a} = \tilde{a'} \).
Properties 3, 6, and 8 are trivial, since because \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \).

Now we show the same for servers receiving a request of low sync integrity.

Lemma 33 (Low Sync Integrity Server Request). Let \( S = (D, \phi, t) \) and \( S' = (D', \phi', t') \) with \( S \Rightarrow S' \) and \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \) \( \alpha = req(t_b, n, u, p, c, k, \omega) \)^{\ell_a,\omega} \( \Gamma \models_{\ell_a,\omega} S \) and \( \Gamma \models_{\ell_a,\omega} \alpha \). Then \( S \models_{I} S' \).

Proof. We show that all the properties of definition 18 are fulfilled. Properties 2a, 2b, 2c are trivial since \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \). Properties 3 and 4 are trivial with \( D = (D_{I}, D_{b}) \) \( D' = (D'_{I}, D'_{b}) \) we have \( D_{b} = D'_{b} \) and \( \phi = \phi' \). Property 1 follows immediately from lemma 17.

Now we show the same for servers sending a response of low sync integrity.

Lemma 34 (Low Sync Integrity Server Response). Let \( S = (D, \phi, t) \) and \( S' = (D', \phi', t') \) with \( S \Rightarrow S' \) and \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \) \( \alpha = res(t_b, n, u, p, c, k, \omega) \)^{\ell_a,\omega} \( \Gamma \models_{\ell_a,\omega} S' \). Then \( S \models_{I} S' \).

Proof. Then the event \( \alpha \) was produced using rule (S-REPLY) or (S-REDIR). In both cases properties 2a, 2b, 2c, 3 and 4 are trivial since \( E = E' \), \( D = D' \), \( l = l' \) and \( \phi = \phi' \). Property 1 follows immediately from lemma 18.

Finally, we use the previous lemmas to show that if a web system takes a step of low sync integrity, then the state before and after the step are in the relation.

Lemma 35 (Low Sync Integrity Steps). Let \( A, A' \) be web systems with \( A \Rightarrow A' \) for some \( \alpha \) with \( I(\ell_a) \subseteq I \) \( I(\ell_a') \subseteq I \). Then \( A \models_{I} A' \).

Proof. We perform an induction on the rule used to derive the step \( \alpha \).

- (A-NIL) Then, if the step is derived through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction.
- The claim for internal browser steps follows from lemma 29. The claim for internal server steps follows from lemma 30.
- (A-BROWSERSERVER) Then we know by lemma 31 that the browser relation is preserved and that the server step is of low integrity. By lemma 7 we know that the request is well typed. The claim for the server relation then follows from lemma 33.
- (A-SERVERBROWSER) Then by lemma 18 we get that the response is well typed. By lemma 31 we hence know that the browser relation is preserved and that the server step is of low integrity. Then we know by lemma 34 that the server relation is preserved.
- (A-TIMEOUTSEND) Then the claim follows from lemma 31.
- (A-TIMEOUTRECV) Then the claim follows from lemma 32.
- (A-BROATK) Then the claim follows from lemma 31 for the browser step.
- (A-ATKSER) Then the claim follows from lemma 33 for the server step, using lemma 21.
- (A-SERATK) Then the claim follows from lemma 34 for the server step.
- (A-ATKBRO) Then the claim follows from lemma 32 for the browser step, using lemma 22.

We now define the next high integrity state of a deterministically terminating web system as the state that is just before processing the next event with high sync integrity. This state can be reached by processing a number of events with low sync integrity. We furthermore show that

1) this state is unique
2) The web system before and after taking the steps with low sync integrity are in the relation.
3) The web system in the newly reached state is still deterministically terminating
4) The web system has a special form (one of the few specified in the lemma)

Lemma 36 (Low Integrity Catch Up). Let \( A = (K, \ell_a) \Rightarrow W \) be a deterministically terminating web system. We say that it is in a low integrity state if:

- \( \{ B \} = \text{browsers}(W) \), \( B = (N, M, P, T, Q, \tilde{a})^{\ell_a,\tilde{a}} \) and \( I(\ell_a) \subseteq I \).
- or there is a \( t \in \{ t' \mid S \in \text{servers}(W) \land t' \in \text{running}(S) \} \) with \( t = [c]^{\ell_a,\tilde{a}} \).

We now define the next high integrity state of a deterministically terminating web system as the state that is just before processing the next event with high sync integrity. This state can be reached by processing a number of events with low sync integrity. We furthermore show that

1) this state is unique
2) The web system before and after taking the steps with low sync integrity are in the relation.
3) The web system in the newly reached state is still deterministically terminating
4) The web system has a special form (one of the few specified in the lemma)

Lemma 36 (Low Integrity Catch Up). Let \( A = (K, \ell_a) \Rightarrow W \) be a deterministically terminating web system. We say that it is in a low integrity state if:

- \( \{ B \} = \text{browsers}(W) \), \( B = (N, M, P, T, Q, \tilde{a})^{\ell_a,\tilde{a}} \) and \( I(\ell_a) \subseteq I \).
- or there is a \( t \in \{ t' \mid S \in \text{servers}(W) \land t' \in \text{running}(S) \} \) with \( t = [c]^{\ell_a,\tilde{a}} \).
We then let $\text{nexth}(A)$ be the websystem $A' = (K', \ell_a) \triangleright W'$ such that

- $A \rightarrow B'$ with $I(\ell_a) \subseteq I \text{ sync}_1(\beta)$ for all $\beta \in \bar{\beta}$.
- for all $A'$, $\alpha$ with $A' \rightarrow A'' \text{ } I(\ell_a) \subseteq I \text{ sync}_1(\alpha)$

We then know that

1) There exists such a unique $A'$
2) $A \equiv \circlearrowleft A'$
3) $A'$ is deterministically terminating

4) Let $B = (N, M, P, T, Q, \vec{a}), \mu$ be the honest browser in $W$ and let $B'$ be the honest browser in $W'$. Then exactly one of the following claims holds:
   a) $B' = (\{\}, M', P', \{\text{tab} \rightarrow \text{skip}\}, \{\}, \vec{a'})^\text{usr,l,μ}$
   b) there exists a server $S$ in $W'$ with $t \in \text{running}(S)$ and $t = [\text{reset } l''; c]^{l'' | l''}_{R,E} \text{ and } I(\ell_a) \subseteq I l''$
   c) $B' = (N', M', P', T', \{\}, \vec{a'})^\text{usr,l,μ}$ with $N = \{n \mapsto \_\}$, $I(\ell_a) \subseteq I l'$ and there exists a server $S$ in $W'$ with $t \in \text{running}(S)$ and $t = [\text{reply } (\text{page}, s, \text{ck}) \text{ with } s \vec{e}]^{l' | l'}_{R,E} \text{ and } R = n \downarrow l''$
   d) $B' = (N', M', P', T', \{\}, \vec{a'})^\text{usr,l,μ}$ with $N = \{n \mapsto \_\}$, $I(\ell_a) \subseteq I l'$ and there exists a server $S$ in $W'$ with $t \in \text{running}(S)$ and $t = [\text{redirect } (u, p, \text{ck}) \text{ with } s \vec{e}]^{l' | l'}_{R,E} \text{ and } R = n \downarrow l''$
   e) $B' = (N', M', P', T', \{\}, \vec{a'})^\text{usr,l,μ}$ with $N = \{n \mapsto \_\}$, $I(\ell_a) \subseteq I l'$ and $T'_O = \{\_ n \downarrow l''\}$

Proof. We show that the different claims hold:

1) The existence and uniqueness follow immediately from the fact the $W$ is deterministically terminating, using definition 17
2) $A \equiv \circlearrowleft A'$ follows from repeated application of lemma 25 and lemma 28
3) Deterministic termination for $W'$ follows immediately from deterministic termination of $W$, using definition 17
4) The form of $W'$ follows from the observation that these five points are the only ones in the semantic rules, where the integrity is raised.

Next, we show that if two high integrity browsers are in the relation and the left browser takes an internal step of high sync integrity, then also the right browser can take the same step and the resulting browsers are still in the relation.

Lemma 37 (High Sync Integrity Browser Steps). Let $B_1 = (N, M, P, T, Q, \vec{a})^\text{usr,l,μ}$ and $B_2 = (N', M', P', T', Q', \vec{a'})^\text{usr,l,μ}$ be browsers with $B_1 \approx B_2$ and $I(\ell_a) \subseteq I l$ and let $B_1 \cdot [\text{some}] \rightarrow B_1'$ with $I(\ell_a) \subseteq I l$. Then there exist $B_2'$ such that $B_2 \cdot [\text{some}] \rightarrow B_2'$ and $B_1 \approx B_2'$.

Proof. We show that all properties of definition 16 are fulfilled. In all cases property 6 follows immediately from lemma 6

Because of $B_1 \approx B_2$ we know

- $\Gamma \vdash_{\ell_a, \text{usr}} B$ and $\Gamma \vdash_{\ell_a, \text{usr}} B'$
- $l = l'$
- $N = N'$
- $K =_{\Gamma, l} K'$
- $\text{dom}(T) = \text{dom}(T')$ and if $T = \{t \mapsto s\}$ and $T' = \{t \mapsto s'\}$ then $s =_{\downarrow l} s'$
- $\vec{a} = \vec{a'}$

By property 1 of definition 11 and because of $I(\ell_a) \subseteq I l$ we know that $\mu$ is hon.

We perform an induction on the derivation of the step $\alpha$.

- (B-SEQ) The claim follows by induction.
- (B-SKIPI) Trivial because $s =_{\downarrow l} s'$
- (B-END) Trivial because $s =_{\downarrow l} s'$
- (B-SETRERE) Then because of $N = N'$, $s =_{\Gamma, l} s'$, we can also apply (B-SETRERE) for $B_2$. We have that $s = v \mapsto v$ and $s' = v \mapsto v'$ where $be =_{\downarrow l} be'$ and the claim follows immediately using lemma 25
- (B-SETDOM) Then because of $N = N'$, $s =_{\Gamma, l} s'$, we can also apply (B-SETDOM) for $B_2$. We have that $s = \text{setdom}(be, u, be'$ and $s' = \text{setdom}(be, u, be')$ where $be =_{\downarrow l} be'$ and $\forall k \in \{1 \ldots |be|\}, be_k =_{\downarrow l} be_k$. By rule (T-BSEDOM) we know that $be = v^\tau$ and $be' = v'^\tau$ are primitive values with $I(\tau) = I(\tau') = \downarrow l$. Hence we know $v = v'$ by the definition of $\approx$. Using lemma 25 for all expressions in $be$, we get page = $\downarrow l$ page' and the claim follows.
- (B-LOAD) Because of $N = N'$, $\text{dom}(T) = \text{dom}(T')$, and $\vec{a} = \vec{a'}$, we can also apply rule (B-LOAD) in $B_2$
All properties except for property \text{6} and \text{8} are trivial.

For simplicity we assume that the names \( n \) and \( n' \) sampled in the two browser are the same, i.e., we have \( n = n' \), and property \text{3} follows immediately.

For property \text{8} the only non-trivial condition is the claim on the cookies of the produced event. This however follows immediately from \( K = \Gamma, \alpha, \bot, 1' \).

- \text{(B-INCLUDE)} Because of \( N = N', \) we can also apply (B-INCLUDE) for \( B_2 \). For simplicity we assume that the names \( n \) and \( n' \) sampled in the two browser are the same, i.e., we have \( n = n' \), and property \text{3} follows immediately using property \text{4} to get that the DOM is of high integrity and hence the origins of the two requests are the same. For property \text{8} the only non-trivial conditions are the claim on the parameters and the cookies of the produced event. These however follow immediately from \( s = \bot, s' \) using lemma \text{25} and \( K = \Gamma, \alpha, \bot, 1' \).

- \text{(B-SUBMIT)} Because of \( N = N', \) dom\((T) = \) dom\((T')\), and \( \vec{a} = \vec{a}' \). Hence we can also apply (B-SUBMIT) in \( B_2 \) and all properties except for property \text{3} and \text{8} follow immediately. Let \( l_D \) be the integrity label of the DOM. We distinguish two cases:
  - \( I(\ell_a) \nsubseteq I \) Then property \text{8} follows immediately. For property \text{8} the only non-trivial conditions are the claim on the parameters and the cookies of the produced event. The claim on the parameters follows directly from \( = \bot, \) on the DOM and \( \vec{a} = \vec{a}' \) and the claim of the cookies follows immediately from \( K = \Gamma, \alpha, \bot, 1' \).
  - \( I(\ell_a) \subseteq I \) Then the claim is trivial.

\( \square \)

Next, we show the same property for high integrity browsers receiving a response of high sync integrity.

\textbf{Lemma 39} (High Sync Integrity Browser Request). Let \( B_1 = (N, M, P, T, Q, \vec{a})^{\text{usr}, \alpha, \mu} \) and \( B_2 = (N', M', P', T', Q', \vec{a}')^{\text{usr}, \alpha', \mu'} \) be browsers with \( B_1 \approx_B B_2 \) and let \( B_1 \xrightarrow{\alpha} B_1' \) with \( I(\ell_a) \nsubseteq I \) sync\(_I\)(\(\alpha\)) and \( \alpha = \text{res}(\ell_b, n, u, p, c, k, o) I^{\text{usr}, \alpha, \mu} \). Then there exist \( B_2' \) and \( \alpha' \) such that \( B_2 \xrightarrow{\alpha'} B_2' \) and \( \alpha = \bot, \alpha' \) and \( B_1 \approx_B B_2 \).

\textbf{Proof.} We know that rule (B-FLUSH) was used and we know that \( Q = \{\alpha\} \).

We then know by \( B_1 \approx_B B_2 \) that if \( Q = \{\alpha'\} \) with \( = \bot, \alpha' \).

We can thus also apply rule (B-FLUSH) in \( B_2 \) and all claims follows immediately.

\( \square \)

Next we show the same property for high integrity browsers receiving a response of high sync integrity.

\textbf{Lemma 39} (High Sync Integrity Browser Request). Let \( B_1, B_2 \) be browsers with \( B_1 \approx_B B_2 \) and let \( B_1 \xrightarrow{\alpha} B_1' \) with \( I(\ell_a) \nsubseteq I \) sync\(_I\)(\(\alpha\)) and \( \alpha = \text{res}(\ell_b, n, u, p, c, k, o) I^{\text{usr}, \alpha, \mu} \). Then there exist \( B_2' \) and \( \alpha' \) such that \( B_2 \xrightarrow{\alpha'} B_2' \) and \( B_1 \approx_B B_2' \).

\textbf{Proof.} We show that all properties of definition \text{10} are fulfilled.

In all cases property \text{6} follows immediately from lemma \text{8}.

Let \( B_1 = (N, M, P, T, Q, \vec{a})^{\text{usr}, \alpha, \mu} \) and \( B_2 = (N', M', P', T', Q', \vec{a}')^{\text{usr}, \alpha', \mu'} \).

Because of \( I(\ell_a) \nsubseteq I \) then we know \( I(\ell_a) \nsubseteq I \) since the integrity label can not be raised between the request and the response, and \( I(\ell_a) \subseteq I \) inspection of the possible rules.

We perform a case distinguish the three possible rules.

- \text{(B-RECVLOAD)} Then because of \( N = N', \) dom\((T) = \) dom\((T')\), we can also apply (B-RECVLOAD) for \( B_2 \). We get \( B_1' \approx_B B_2' \) from \( = \bot, \alpha' \).

- \text{(B-RECVINCLUDE)} Then \( T = \{\text{tab} \mapsto s\} \) and \( T' = \{\text{tab} \mapsto s'\} \) Then because of \( N = N', \) \( = \bot, s' \), we can also apply (B-RECVINCLUDE) for \( B_2 \). We get \( B_1' \approx_B B_2' \) from \( = \bot, \alpha' \).

- \text{(B-REDIRECT)} Then because of \( N = N', \), we can also apply (B-REDIRECT) for \( B_2 \). We get \( B_1' \approx_B B_2' \) from \( = \bot, \alpha' \) and \( K = \Gamma, \alpha, 1' \).

\( \square \)

The next lemma treats the case, where a browser receives a response that is of high sync integrity, but of low integrity.

\textbf{Lemma 40} (High Sync Integrity Browser Response of Low Integrity). Let \( B_1, B_2 \) be browsers with \( B_1 \approx_B B_2 \) and let \( B_1 \xrightarrow{\alpha} B_1' \) with \( I(\ell_a) \nsubseteq I \) sync\(_I\)(\(\alpha\)) and \( \alpha = \text{res}(\ell_b, n, u, p, c, k, o) I^{\text{usr}, \alpha, \mu} \) with \( I(\ell_a) \subseteq I \) \( = \bot, \alpha' \), \( \Gamma = \bot, \alpha \) and \( = \bot, \alpha' \). Then there exist \( B_2' \) and such that \( B_2 \xrightarrow{\alpha'} B_2' \) and \( B_1 \approx_B B_2' \).

\textbf{Proof.} We show that all properties of definition \text{10} are fulfilled.

In all cases property \text{6} follows immediately from lemma \text{8}.

Let \( B_1 = (N, M, P, T, Q, \vec{a})^{\text{usr}, \alpha, \mu} \) and \( B_2 = (N', M', P', T', Q', \vec{a}')^{\text{usr}, \alpha', \mu'} \).
We perform a case distinction between the three possible rules.

- (B-RECVLOAD) Then we know that $I(\ell_a) \not\subseteq I \ell$. Then because of $B_1 \approx B_2$ we get that $N = N'$, $\text{dom}(T) = \text{dom}(T')$ and we can also apply (B-RECVLOAD) or (B-REDIRECT) for $B_2$. Because of $I(\ell_a) \not\subseteq I \ell$ we immediately get $B_1 \approx B_2$.

- (B-RECVINCLUDE) Then we know that $I(\ell_a) \not\subseteq I \ell$. As a high integrity script cannot receive a low integrity response, this case is impossible.

- (B-REDIRECT) Then we know that $I(\ell_a) \not\subseteq I \ell$. Then because of $B_1 \approx B_2$ we get that $N = N'$, $\text{dom}(T) = \text{dom}(T')$ and we can also apply (B-RECVLOAD) or (B-REDIRECT) for $B_2$. Because of $I(\ell_a) \not\subseteq I \ell$ we immediately get $B_1 \approx B_2$. Then because of $N = N'$, we can also apply (B-REDIRECT) or (B-LOAD) for $B_2$.

Next we show the same property for servers taking an internal step of high sync integrity.

**Lemma 41** (High Sync Integrity Server Steps). Let $S_1, S_2$ be servers with $S_1 \approx S_2$ and let $S_2$ be deterministically terminating. Let $S_1 \xrightarrow{\alpha} S_1'$ with $I(\ell_a) \not\subseteq I \ell$ and $\alpha \in \{\cdot, \vdash, \cdot\}$. Then $\text{bad}(S_1')$ or there exist $S_2'$ and $\alpha'$, $\beta$ such that $S_2 \xrightarrow{\alpha', \beta} S_2'$ with $I(\ell_a) \not\subseteq I \ell$ and by the definition of $\approx$ we know that there exists a corresponding thread $c(t_1) = t_2 \in \text{running}(S_2')$ with $t_2 = [c_2', \phi_2', t_2']$. We now show that there are $\alpha, \beta$ and $t_2' = [c_2', \phi_2', t_2']$ with $I(\ell_a) \not\subseteq I \ell$ and the claim is trivial.

• (S-SEQ) Claim follows by induction.

• (S-IFTRUE) Then $c_1 = \text{if } s \text{ then } c_{11} \text{ else } c_{12}$ and $c_2 = \text{if } s' \text{ then } c_{21} \text{ else } c_{22}$ with $s =_{\ell} s'$. Let $v^\tau = \text{eval}_{E_1}(s, D_1)$ and $v'^\tau = \text{eval}_{E_2}(s', D_2)$. Then by lemma 27 we get $v^\tau =_{\ell} v'^\tau$. We distinguish two cases:
  - If reply, redirect, tokencheck, origincheck $\in \text{cons}(c_{11}) \cup \text{cons}(c_{12})$. We distinguish to cases
    * If $I(\ell_a) \not\subseteq I \ell$ then we also have $I(\ell_a) \not\subseteq I \ell. \text{ if } v = v'$. Hence the continuations are $c_1 = c_{11}$ and $c_2 = c_{12}$ and the claim follows immediately.
    * If $I(\ell_a) \subseteq I \ell. \text{ if } v = v'$. We have $I(\ell_a) \subseteq I \ell. \text{ if } v = v'$. Then $c_1 = [c_{11}; \text{reset } l_1]; E_1, R_1$, $c_2 = [c_{12}; \text{reset } l_2]; E_2, R_2$.
  - If reply, redirect, tokencheck, origincheck $\notin \text{cons}(c_{11}) \cup \text{cons}(c_{12})$. We distinguish to cases
    * If $I(\ell_a) \not\subseteq I \ell$ then we also have $I(\ell_a) \not\subseteq I \ell. \text{ if } v = v'$. Hence the continuations are $c_1 = \text{reset } l_1$ and $c_2 = \text{reset } l_2$ and the claim follows immediately.

• (S-FALSE) This case is analog to the case of (T-TRUE).

• (S-TOKENCHECKTRUE). Then $c_1 = \text{if } \text{tokencheck}(s_{e1}, s_{e2}) \text{ then } c_{1}'$ and $c_2 = \text{if } \text{tokencheck}(s_{e1}, s_{e2}) \text{ then } c_{2}'$.

Let $v_{11} = \text{eval}_{E_1}(s_{e1}, D_1), v_{12} = \text{eval}_{E_1}(s_{e2}, D_1), v_{21} = \text{eval}_{E_2}(s_{e1}, D_2), v_{22} = \text{eval}_{E_2}(s_{e2}, D_2)$.

We then know that $v_{11} = v_{12}$.

We distinguish two cases:
  - If $v_{21} = v_{22}$ then $c_1 = c_{1}'$ and $c_2 = c_{2}'$ and the claim is trivial.
  - If $v_{21} \neq v_{22}$ then we have $c_2 = \text{reply } (\text{error, skip, } \{\})$. This however is a contradiction to the assumption of the deterministic termination.

• (S-TOKENCHECKFALSE) Then $t_1' = [\text{reply } (\text{error, skip, } \{})]_{E_1, R_1}$ and the claim is trivial, since we have $\text{bad}(S_1')$.

• (S-SKIP) Trivial.

• (S-RESET) We distinguish two cases:
  - If $I(\ell_a) \not\subseteq I \ell$ then we have $I(\ell_a) \not\subseteq I \ell$. Otherwise we know that $c_1 = \text{reset } l_1$ where $I(\ell_a) \not\subseteq I \ell$. Then we know by property 28 of definition 18 that $c_2 = c_{2}{r}$. For some $c_{2'r}$ with $c_1 = l_1, c_{2'r}$.

We then let $c' = c_{2'}$, and show that they fulfill the claim.

Since we know that $\text{reply, redirect, tokencheck, origincheck } \notin \text{cons}(c_{2'}r)$, we know by deterministic termination of $t_2$ that $t_2 = \text{reset } l_2; c'$. By repeated application of lemma 30 and lemma 26 we get $S_1 \approx S_2''$. 

Now we need to show \( S'_1 \approx_\delta S'_2 \). All claims from definition \( \ref{def:sync} \) except for property \( \ref{def:sync:1} \) are trivial. For property \( \ref{def:sync:1} \) \( l'_1 = l'_2 \) follows immediately from rule (S-RESET) and \( c'_1 = _{=1} c'_2 \) follows immediately from \( c'_1 = _{=1} c'_2 \).

- **(S-RESTORESESSION)** Then \( c_1 = \text{start } se \) and \( c_2 = \text{start } se' \), with \( se = _{=1} se' \). Then for \( S_2 \) we can apply rule (S-RESTORESESSION) or (S-NEWSESSION) and the claim follows because using lemma \( \ref{lem:sync_EQ} \) we immediately get \( j'_1 = _{=1} j'_2 \).

- **(S-NEWSESSION)** Analog to previous case.

- **(S-SETGLOBAL)** We have \( c_1 = r := se \) and \( c_2 = r := se' \) with \( se = _{=1} se' \). The claim then follows immediately using lemma \( \ref{lem:sync_EQ} \).

- **(S-SETSESSION)** This case follows analog to the previous one.

- **(S-LOGIN)** We have \( c_1 = \text{login } se_1, se_2, se_3 \) and \( c_2 = \text{login } se'_1, se'_2, se'_3 \) with \( se_1 = _{=1} se'_1, se_2 = _{=1} se'_2 \) and \( se_3 = _{=1} se'_3 \). Let \( j'_1 = \text{eval}_{E_1}(se_3, D_1) \) and let \( j'_2 = \text{eval}_{E'_2}(se'_3, D_2) \). We distinguish two cases:
  - If \( I(\ell_a) \subseteq I(\tau) \) then also \( I(\ell_a) \subseteq I(\tau') \) and the claim follows immediately.
  - If \( I(\ell_a) \not\subseteq I(\tau) \) then \( j'_1 = j'_2 \). By rule (T-LOGIN) and lemma \( \ref{lem:sync_EQ} \) we know that \( \tau = \text{cred}(\ell) \). Let \( v_{i1} = \text{eval}_{E_1}(se_1, D_1) \), let \( v_{2i} = \text{eval}_{E'_2}(se'_2, D_2) \). Then by rule (T-LOGIN) and lemma \( \ref{lem:sync_EQ} \) we know that and \( I(\ell_a) \not\subseteq I(\tau_1) \) and \( I(\ell_a) \not\subseteq I(\tau_2) \) and hence by \( se_1 = _{=1} se'_1 \) and \( se_2 = _{=1} se'_2 \) we get \( v_{i1} = v_{r1} \) and \( v_{2i} = v_{r2} \). With \( \tau_0 = \text{eval}_{E_1}(se_1, D_1) \) and let \( \tau' = \text{eval}_{E_2}(se'_1, D_2) \) we get using the properties of \( \rho \).

- **(T-AUTH)** Then we have \( c_1 = \text{auth } se_1 \) \( \ell \) and \( c_2 = \text{auth } se'_2 \) \( \ell \) with \( se_1 = _{=1} se'_2 \). If \( I(\ell_a) \subseteq I(\ell) \), then the claim is trivial. We hence assume \( I(\ell_a) \not\subseteq I(\ell) \).
  - We then know by rule (T-AUTH) that \( I(\ell_a) \not\subseteq I(\ell) \).
  - Let \( v_{1i} = \text{eval}_{E_1}(se_1, D_1) \) and \( v_{2i} = \text{eval}_{E_2}(se_2, D_2) \). Let \( R_1 = R_2 = n, u, u_b, o \), let \( E_1 = _{=1} j_1 \) and \( E_2 = _{=1} j_2 \) and let \( \ell_1 = \phi(j_1) \) and \( \ell_2 = \phi(j_2) \).
  - We then know \( se_1 = _{=1} se'_2 \) and \( j_1 = _{=1} j_2 \).
  - We have \( \alpha = [v_{1i}]_{\ell_1}^{\ell_1} \) and \( \alpha' = [v_{2i}]_{\ell_2}^{\ell_2} \).

By rule (T-AUTH) we know that \( I(\ell_a) \not\subseteq I(\tau_1) \) and \( I(\ell_a) \not\subseteq I(\tau_2) \). We hence have \( v_{i1} = v_{j1} \) by lemma \( \ref{lem:sync_EQ} \).

By rule (T-AUTH) we also know that \( I(\ell_a) \not\subseteq I(\tau_1) \) and \( I(\ell_a) \not\subseteq I(\tau_2) \). We hence get by property \( \ref{def:sync:1} \) of definition \( \ref{def:sync} \) that \( \tau_1 = \tau_2 \).

We thus have \( \alpha = \alpha' \) and the claim follows.

- **(S-OCHECKSUC)** We then have \( c_1 = \text{if originchk}(O) \) then \( c'_1 \) and \( c_2 = \text{if originchk}(O) \) then \( c'_2 \). With \( R_1 = R_2 = n, u, u_b, o \) we know that we can also apply rule (S-OCHECKSUC) in \( t_2 \) and the claim follows immediately.

- **(S-OCHECKFAIL)** We then have \( c_1 = \text{if originchk}(O) \) then \( c'_1 \) and \( c_2 = \text{if originchk}(O) \) then \( c'_2 \). With \( R_1 = R_2 = n, u, u_b, o \) we know that we can also apply rule (S-OCHECKFAIL) in \( t_2 \) which contradicts our assumption about the termination of \( t_2 \). This case is thus impossible.

Next, we show the same property for servers receiving a request of high sync integrity.

**Lemma 42 (High Sync Integrity Server Request).** Let \( S_1, S_2 \) be servers with \( S_1 \approx_\delta S_2 \) and let \( S_2 \) be the corresponding server of \( S_1 \) as defined in definition \( \ref{def:sync} \). Let \( S_1 \rightsquigarrow S'_1 \) with \( I(\ell_a) \not\subseteq I(\tau) \) and \( \alpha = \text{req}(t_b, n, u, p, c, k, o)^{l, \mu} \). Let \( \alpha' \) with \( \alpha = _{=1} \alpha' \) and \( \Gamma \triangleright \ell_a, \text{usr } \alpha, \Gamma \triangleright \ell_a, \text{usr } \alpha' \). Then there exist \( S'_2 \) such that \( S_2 \rightsquigarrow S'_2 \) and \( S'_1 \approx_\delta S'_2 \).

**Proof.** Then the step is taken using rule (S-RECV). We hence have \( t = [v_{i1}]_{(x)} \leftrightarrow c \in \text{threads}(S_1) \). Because \( S_2 \) is the corresponding server of \( S_1 \) we know that \( t \in \text{threads}(S_2) \).

We can thus apply rule (S-RECV) in and take the step \( S'_2 \rightsquigarrow S'_2 \).

We now show \( S'_1 \approx_\delta S'_2 \) by showing the properties of definition \( \ref{def:sync} \).

- Property \( \ref{def:sync:1} \) follows immediately from lemma \( \ref{lem:sync_EQ} \).
- Property \( \ref{def:sync:2} \) is trivial (For simplicity we assume that sampling returns the same result on both servers).
- Property \( \ref{def:sync:3} \) follows immediately from the claim on cookies in \( \alpha = _{=1} \alpha \).
- Property \( \ref{def:sync:4} \) follows from the claim on the parameters in \( \alpha = _{=1} \alpha \).
- Properties \( \ref{def:sync:5} \) and \( \ref{def:sync:6} \) are trivial since the session memory and trust mapping are not modified in rule (S-RECV).

Let \( t \) and \( t' \) be the freshly generated running threads. Then \( t = _{=1} t' \) follows from the claim on \( p \) in \( \alpha = _{=1} \alpha' \).

Next, we show the same property for servers sending a response of high sync integrity.

**Lemma 43 (High Sync Integrity Server Response).** Let \( S_1, S_2 \) be servers with \( S_1 \approx_\delta S_2 \) and let \( S_1 \rightsquigarrow S'_1 \) with \( I(\ell_a) \not\subseteq I(\tau) \) and \( \alpha = \text{req}(t_b, n, u, u', \ell, c, k, p, s)^{l, \mu} \). Then there exist \( S'_2 \) and \( \alpha' \) such that \( S_2 \rightsquigarrow S'_2 \) and \( S'_1 \approx_\delta S'_2 \) and \( \alpha = _{=1} \alpha' \).

**Proof.** We distinguish two cases for the rule applied to take the step:
(S-REPLY) We have \( c = \text{reply} (p, a, c) \) with \( \vec{x} = \vec{s} \) and \( c' = \text{reply} (p, a, c) \) with \( \vec{x} = \vec{s}' \) with \( \forall i \in [1 \ldots |s|] \), \( se_i = \perp, se_i' \).

Let \( v_i = \text{eval}_E (se_i, D) \) and \( v'_i = \text{eval}_E (se_i', D') \). Then by lemma 27 we get \( v_i = \perp, v'_i \).

With \( \sigma = \{ x_1 \rightarrow v_1 \ldots x_m \rightarrow v_m \} \) and \( \sigma' = \{ x'_1 \rightarrow v'_1 \ldots x'_m \rightarrow v'_m \} \), we immediately get \( sa = \perp, sa', page\sigma = \perp, page\sigma' \) and \( ck\sigma = \perp, ck\sigma' \).

The claim then follows using lemma 18.

(S-REDIR) We have \( c = \text{redirect} (u, \vec{z}, c) \) with \( \vec{x} = \vec{s} \) and \( c' = \text{redirect} (u, \vec{z}, c) \) with \( \vec{x} = \vec{s}' \) with \( \forall i \in [1 \ldots |s|] \), \( se_i = \perp, se_i' \).

Let \( v_i = \text{eval}_E (se_i, D) \) and \( v'_i = \text{eval}_E (se'_i, D') \). Then by lemma 27 we get \( v_i = \perp, v'_i \).

With \( \sigma = \{ x_1 \rightarrow v_1 \ldots x_m \rightarrow v_m \} \) and \( \sigma' = \{ x'_1 \rightarrow v'_1 \ldots x'_m \rightarrow v'_m \} \), we immediately get \( \vec{z}\sigma = \perp, \vec{z}\sigma' \) and \( \vec{ck}\sigma = \perp, \vec{ck}\sigma' \).

The claim then follows using lemma 18.

Finally, we show the same property on web system level.

**Lemma 44 (High Sync Integrity Steps).** Let \( A_1 = (K_1, \ell_a) \models W_1 \) and \( A_2 = (K_2, \ell_a) \models W_2 \) be web systems with \( A_1 \approx_1 A_2 \) and let \( A_2 \) be deterministically terminating. Then whenever \( A_1 \xrightarrow{\alpha} A'_1 \) with \( I(\ell_a) \nsubseteq_1 \text{sync}_1(\alpha) \) then \( \exists \beta, \alpha' \) such that \( A_2 \xrightarrow{\beta} A'_2 \) with \( \alpha = \perp, \alpha' \) and for all \( \beta \in \beta \) we have \( I(\ell_a) \subseteq_1 \text{sync}_1(\beta) \) and \( A'_1 \approx_1 A'_2 \).

**Proof.** If \( A_2 \) is not in a low integrity state as defined in lemma 56 then let \( A'_2 = A_2 \). Otherwise, let \( A'_2 = \text{next} (A_2) \) as in lemma 56. We then know that \( A_2 \xrightarrow{\beta} A'_2 \) where for \( \beta \in \beta \) we have \( I(\ell_a) \subseteq_1 \text{sync}_1(\beta) \) and \( A'_2 \approx_1 A'_2 \). By transitivity we hence get \( A_1 \approx_1 A'_2 \). We furthermore know that \( A'_2 \) is in one of the five states described in lemma 56.

We now show that \( A'_2 \xrightarrow{\alpha} A'_2 \) with \( \alpha = \perp, \alpha' \) and \( A_2 \approx_1 A'_2 \). We prove the claim by induction over the derivation of the step \( \alpha \).

(A-NIL) Then, if the step is derived through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction. The claim for internal server steps follows from lemma 51. For internal browser steps, we perform a case distinction: Let browsers(\( W \)) \models B = (N, K, P, T, Q, \vec{a})_{\text{arr}, \mu}.

- if \( I(\ell_a) \nsubseteq_1 l \) then the claim follows immediately from lemma 57.

- if \( I(\ell_a) \subseteq_1 l \) then we know that rule (B-END) is used. We know that \( A'_2 = \text{next} (A_2) \) is in one of the five states described in lemma 56. Since we already excluded one possible state, and three other states require \( I(\ell_a) \subseteq_1 l \), we know that the browser \( B_2'' \) in \( A'_2 \) is in a state where rule (B-END) can be used. The claim then follows immediately.

(A-BROWSERSERVER) Then we can also apply (A-BROWSERSERVER) for \( A''_2 \) and the claim follows from lemma 38 for the browser step and lemma 42 for the server in case of a high integrity request or lemma 33 for the server in case of a low integrity request.

(A-SERVERBROWSER) Then we distinguish two cases

- Integrity of the response is high: Then we can also apply rule (A-SERVERBROWSER) in \( A''_2 = A_2 \) and the claim follows from lemma 33 for the server and lemma 39 for the browser step.

- Integrity of the response is low: Then we know by lemma 56 that \( A''_2 = \text{next} (A_2) \) is in a state where the browser can receive a request and the server can send a reply or a redirect or a timeout response is ready to be sent. We immediately get \( \alpha = \perp, \alpha' \). Then we can also apply rule (A-SERVERBROWSER) and the claim follows from lemma 56 for the server step and from lemma 40 for the browser step.

(A-TIMOUTSEND) Then we can also apply (A-TIMOUTSEND) in \( A''_2 \) and the claim follows from lemma 38 for the browser step in case of a high integrity browser state or from lemma 31 in case of a low integrity browser state.

(A-TIMOUTRECV) Then we can also apply (A-TIMOUTRECV) in \( A''_2 \) and the claim follows from lemma 39 or lemma 51 for the browser step.

(A-BROATK) Then we distinguish two cases

- \( W_2 \) can perform a step using (A-BROWSERSERVER): Then the claim follows from lemma 38 or lemma 51 for the browser step and lemma 30 for the server.

- \( W_2 \) can perform a step using (A-TIMOUTSEND) Then the claim follows from lemma 38 or lemma 51 for the browser step.

(A-ATKSER) Cannot happen, event is of high integrity

(A-SERATK) Cannot happen, event is of high integrity

(A-ATKBRO) Then we distinguish two cases:

- \( W_2 \) can perform a step using (A-SERVERBROWSER): Then the claim follows from lemma 59 for the browser step.
W_2 

Using the previous lemmas, we can conclude that the relation \(\cong_{\Gamma} \) fulfills core properties, that will allow us to prove the main theorem.

**Lemma 45.** Let \(A_1 \) and \(A_2 \) be web systems with \(A_1 \cong_{\Gamma} A_2 \). Then

1) \(bad(A_1)\)
2) or the following properties hold:
   a) if \(A_1 \cong_{\Gamma} A'_1 \) and \(I(\ell_a) \not\subseteq I sync_{\Gamma}(\alpha)\) then there exists \(\alpha' , \beta \) and \(A'_2 \) such that
      - for all \(\beta \in \vec{\beta} \) we have \(I(\ell_a) \subseteq I sync_{\Gamma}(\beta)\)
      - \(A_2 \equiv_{\Gamma}^* A'_2\)
      - \(\alpha = I_{\Gamma}^* \alpha'\)
      - \(A'_1 \cong_{\Gamma} A'_2\)
   b) if \(A_1 \cong_{\Gamma} A'_1\) for some \(\alpha \) with \(I(\ell_a) \subseteq I sync_{\Gamma}(\alpha)\) then \(A'_1 \cong_{\Gamma} A_2\).

*Proof.* If \(bad(A_1)\) then the claim is trivial. Hence assume \(\neg bad(A_1)\), which then immediately gives us \(A_1 \cong_{\Gamma} A_2\). The claim for the low integrity step then follows immediately from lemma [35](#) and the transitivity of \(\cong_{\Gamma} \) (lemma [28](#)) and the claim for the high integrity step follows from lemma [46](#).

Intuitively, the relation fulfills the following properties: Either the first websystem is in a bad state, or

1) Whenever the first system takes a step of high sync integrity, then the second system can take a number of steps of low sync integrity, followed by the same step of high sync integrity, and the resulting websystems are in the relation.

2) If the first system takes a step of low sync integrity, then it remains in relation with the second system (which didn’t take a step).

**L. Main Theorem**

In this section we bring together the results from the previous sections in order to show our main theorem.

First, we show that whenever an attacked and an unattacked websystem are in the relation \(\cong_{\Gamma} \), and the attacked system generates a trace, then the unattacked websystem can generate a trace that has the same events with high sync integrity.

**Lemma 46 (High Integrity Trace Equality).** Let \(high(\gamma)\) be the trace containing only the events \(\alpha \neq \bullet \) with \(I(\ell_a) \not\subseteq I sync_{\Gamma}(\alpha)\).

Let \(A_1 \) be an attacked and \(A_2 \) an unattacked websystem with \(A_1 \cong_{\Gamma} A_2\). Then if \(A_1\) generates the trace \(\gamma_1\), then \(A_2\) can generate a trace \(\gamma_2\) such that \(high(\gamma_1) = high(\gamma_2)\).

*Proof.* We prove the claim by induction over the generated trace \(\gamma\), using the properties of \(\cong_{\Gamma} \) from lemma [45](#)

1) If \(\gamma_1 = \epsilon\) then the claim is trivially fulfilled.

2) If \(\gamma_1 = \alpha \cdot \gamma'_1\) then \(A_1\) takes the step \(\alpha\) to reach state \(A'_1\) and produces the trace \(\alpha \cdot \gamma'_1\). If \(bad(A_1)\) then we know that \(high(\alpha \cdot \gamma_1) = \epsilon\), since according to definition \(15\) we either have \(\alpha = \bullet\) or \(I(\ell_a) \subseteq I init(\alpha)\) and the claim is trivial. We hence assume \(\neg bad(A_1)\) and hence know \(A_1 \cong_{\Gamma} A_2\).

We distinguish two cases

a) If \(I(\ell_a) \not\subseteq I sync_{\Gamma}(\alpha)\) then by lemma [25](#) \(A_2\) can take the steps \(\vec{\beta} \cdot \alpha\), where \(I(\ell_a) \subseteq I sync_{\Gamma}(\beta)\) for all \(\beta \in \vec{\beta}\) and hence produces the trace \(\vec{\beta} \cdot \alpha \cdot \gamma'_2\). We hence have \(high(\vec{\beta} \cdot \alpha) = \alpha\). Since \(A'_1 \cong_{\Gamma} A'_2\), we can apply the induction hypothesis and get \(high(\gamma'_1) = high(\gamma'_2)\), hence we also have \(high(\gamma_1) = high(\alpha \cdot \gamma'_1) = high(\alpha \cdot \gamma'_2) = high(\gamma_2)\).

b) If \(I(\ell_a) \subseteq I sync_{\Gamma}(\alpha)\) then we have \(high(\gamma'_1) = high(\gamma_1)\) and since by lemma [45](#) we know \(A'_1 \cong_{\Gamma} A_2\), we can apply the induction hypothesis and get \(high(\gamma_2) = high(\gamma'_1) = high(\gamma_1)\).

Next, we show that whenever a well-typed websystem produces a high integrity authenticated event then it also has high sync integrity.

**Lemma 47 (High Integrity Auth Events).** Let \(usr\) be the honest user and for all \(u\) with \(\rho(usr,u) = n^T\) we have \(C(\tau) \not\subseteq C(\ell_a)\). Let \(\ell_a\) be an attacker. For any \(A\) with \(\Gamma \vdash_{\ell_a,usr} A\) and \(A \overset{\alpha}{\rightarrow} A'\), if for \(\beta = \ell[\vec{\sigma}]^{1,n}\) we have \(\beta \in \vec{\sigma}\) and \(I(\ell_a) \not\subseteq I(\ell)\) and \(\ell = usr\) or \(\ell_a = usr\) then we have \(I(\ell_a) \not\subseteq I sync_{\Gamma}(\beta)\).

*Proof.* Let \(l = sync_{\Gamma}(\beta)\).

There exist \(A_1 , A_2 \) such that \(A \overset{\alpha}{\rightarrow} \cdot A_1 \cdot \beta \rightarrow A_2 \overset{\alpha}{\rightarrow} A'\) with \(\Gamma \vdash_{\ell_a,usr} A_1\) by lemma [23](#)

We also know that \(S_1 \in servers(A_1), S_2 \in servers(A_2)\) with \(S_1 \beta \rightarrow S_2\), with \(\Gamma \vdash_{\ell_a,usr} S_1\) by definition [13](#)
Let $S_1 = (D, \phi, t)$. Then there is $[c]_{\overline{R,E}} \in \text{running}(S_1)$ with $(D, \phi, [c]_{\overline{R,E}}) \xrightarrow{\beta} (D, \phi, [c']_{\overline{R,E}})$ for some $c, c'$.

We know that the event $\beta$ is produced using rule (S-AUTH), hence have

$$(D, \phi, [\text{auth } s \bar{c} \text{ at } \ell]_{\overline{R,E}}) \xrightarrow{\beta} (D, \phi, [\text{skip}]_{\overline{R,E}})$$

with $R = n, u, t_b, o$ and $E = i, j$ with $\phi(j) = t_u$.

Let

$$b = \begin{cases} 
\text{att} & \text{if } t_b \neq \text{usr} \\
\text{hon} & \text{if } \mu = \text{hon} \land t_b = \text{usr} \\
\text{csrf} & \text{if } \mu = \text{att} \land t_b = \text{usr} 
\end{cases}$$

$$\Gamma' = \begin{cases} 
\Gamma_{\overline{R,a}} & \text{if } (t_b = \text{usr}) \\
\{ \_ \rightarrow \ell_a \} & \text{if } (t_b \neq \text{usr}) 
\end{cases}$$

$$\Gamma'_0 = (\Gamma_0, \Gamma', (\Gamma'_0, \Gamma'_0), \Gamma'_0, \Gamma'_0)$$

We then get by rule (T-RUNNING)

$$\Gamma', j\text{label}(j), l \vdash_{\ell_a, (u,b,p)} \text{auth } s \bar{c} \text{ at } \ell : _{-}l$$

We distinguish two cases:

- If $b \neq \text{att}$ then by typing we know from rule (T-AUTH) that we have $I(\ell_a) \not\subseteq I l$ and the claim follows immediately.
- If $b = \text{att}$ then we know that $t_b \neq \text{usr}$. Hence we must have $\phi(j) = t_u = \text{usr}$. We now show that this case can also not happen. Because of $C(\rho(\ell_a)) \not\subseteq C (\ell_a)$ and property $b$ of definition 12, we know that $C(\rho(\ell_a)) \not\subseteq C (\text{label}(j))$. Since an attacker can never have a session with a high confidentiality session label, we know $C(\text{label}(j)) \not\subseteq C (\ell_a)$ and we immediately have a contradiction. 

\[\Box\]

We define a well-formed attacker to be an attacker whose knowledge is limited by his label.

**Definition 20** (Well-formed attacker). An attacker $(\ell_a, \mathcal{K})$ is well-formed if $\forall \ell \subseteq \ell_a \ell_a$.

This lemma shows that the initial state is in the relation $\approx_{\Gamma}$ with itself.

**Lemma 48** (The initial state is in $\approx_{\Gamma}$). Assume a well-formed server cluster $W_0$, an honest browser of the user $\text{usr} B_{\text{usr}}(\{\}, \bar{a})$ with well formed user actions $\bar{a}$, a well-formed attacker $(\ell_a, \mathcal{K})$ and let $\mathcal{A} = (\mathcal{K}, \ell_a) \triangleright B_{\text{usr}}(\{\}, \bar{a}) \parallel W_0$. If for all servers $S = (D, \phi, t)$, we have $\Gamma_0 \vdash_{\ell_a, B_{\text{usr}}} t$, then $\mathcal{A} \approx_{\Gamma} \mathcal{A}^\triangleright$.

**Proof.** We first show $\mathcal{A} \approx_{\Gamma} \mathcal{A}$ by showing the different properties of definition 19:

- For property 1 $\Gamma \vdash_{\ell_a, \text{usr}} A$ we show that the properties of definition 14 are fulfilled:
  - We get property 1 $\Gamma \vdash_{\ell_a, \text{usr}} B_{\text{usr}}(\{\}, \bar{a})$ by checking that all the properties of definition 11 hold. Property 1 follows from the well-formedness of $\bar{a}$. All other properties are trivial.
  - We get property 2 $\Gamma \vdash_{\ell_a, \text{usr}} S$ for all servers $S = (D, \phi, t) \in \text{servers}(W_0)$ by checking that all properties of definition 12 hold. Property 2 follows from $\Gamma_0 \vdash_{\ell_a, B_{\text{usr}}} t$, using lemma 2. All other properties are trivial for fresh servers (as defined in definition 5).
  - Property 3 is trivial since there are no network connections in the browser.
  - Property 4 follows immediately from the well-formedness of the attacker.

- property 2 is trivial
- For property 3 $\mathcal{A} \approx_{\Gamma} \mathcal{A}$ we show that the properties of definition 18 hold:
  - property 1 follows immediately from $\Gamma \vdash_{\ell_a, \text{usr}} \mathcal{A}$, which we have already shown.
  - All other properties are trivial for fresh servers.

- For property 5 $B_{\text{usr}}(\{\}, \bar{a}) \approx_{\Gamma} B_{\text{usr}}(\{\}, \bar{a})$ we show that the properties of definition 16 hold:
  - property 1 follows immediately from $\Gamma \vdash_{\ell_a, \text{usr}} \mathcal{A}$, which we have already shown.
  - All other properties are trivial for fresh servers.

\[\Box\]

Our main theorem states that typing ensures web session integrity – if we consider all ingredients to be well-formed.

**Theorem 2** (Typing implies Web Session Integrity). Let $W$ be a fresh cluster, $(\ell_a, \mathcal{K})$ a well-formed attacker, $\Gamma^0$ a typing environment with $\lambda, \ell_a, \Gamma^0 \vdash \phi$ and let $\bar{a}$ be a list of well-formed user actions for $\text{usr}$ in $W$ with respect to $\Gamma^0$ and $\ell_a$. Assume
that for all \( u \) with \( \rho(\text{usr}, u) = n^\top \) we have \( C(\tau) \not\subseteq C(\ell_a) \) and that we have \( \Gamma^0 \vdash \ell_a, P \ t \) for all servers \( S = (\{\}, \{\}, t) \) in \( W \). Then \( W \) preserves session integrity against \( (\ell_a, K) \) for the honest user \( \text{usr} \) performing the list of actions \( \vec{a} \).

Proof. Let \( W' = B_{\text{usr}}(\{\}, \vec{a}) \parallel W \) and let \( A = (\ell_a, K) \triangleright W' \).

We have to show that for any attacked trace \( \gamma \) generated by the attacked system \( A \) there exists a corresponding unattacked trace \( \gamma' \) generated by \( A \) such that

\[
\forall I(\ell) \not\subseteq I(\ell') : \gamma \downarrow (\text{usr}, \ell') = \gamma' \downarrow (\text{usr}, \ell')
\]

By lemma lemma 48, we know \( A \cong \overline{A} \).

By lemma lemma 1 we know that also \( A \) can produce the trace \( \alpha \).

Applying lemma 46, we know that there exists an unattacked trace \( \gamma' \) produced by \( \overline{A} \), such that \( \text{high}(\gamma) = \text{high}(\gamma') \).

Since for all \( \alpha = \sharp[\vec{\iota}]; \vec{\iota} \not\subseteq \ell \) we have \( \text{int}(\alpha) = \bot_I \) by lemma 47, we know that

\[
\forall I(\ell) \not\subseteq I(\ell') : \gamma \downarrow (\text{usr}, \ell') = \gamma' \downarrow (\text{usr}, \ell')
\]

By lemma lemma 1 we know that this trace \( \gamma' \) can also be produced by \( A \). 

\( \square \)