

# An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System\*

Daniel Fett, Ralf Küsters, Guido Schmitz  
University of Trier, Germany  
{fett,kuesters,schmitzg}@uni-trier.de

## Abstract

The web constitutes a complex infrastructure and as demonstrated by numerous attacks, rigorous analysis of standards and web applications is indispensable.

Inspired by successful prior work, in particular the work by Akhawe et al. as well as Bansal et al., in this work we propose a formal model for the web infrastructure. While unlike prior works, which aim at automatic analysis, our model so far is not directly amenable to automation, it is much more comprehensive and accurate with respect to the standards and specifications. As such, it can serve as a solid basis for the analysis of a broad range of standards and applications.

As a case study and another important contribution of our work, we use our model to carry out the first rigorous analysis of the BrowserID system (a.k.a. Mozilla Persona), a recently developed complex real-world single sign-on system that employs technologies such as AJAX, cross-document messaging, and HTML5 web storage. Our analysis revealed a number of very critical flaws that could not have been captured in prior models. We propose fixes for the flaws, formally state relevant security properties, and prove that the fixed system in a setting with a so-called secondary identity provider satisfies these security properties in our model. The fixes for the most critical flaws have already been adopted by Mozilla and our findings have been rewarded by the Mozilla Security Bug Bounty Program.

---

\*An abridged version appears in S&P 2014 [14].

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Communication Model</b>	<b>5</b>
2.1	Terms, Messages and Events . . . . .	6
2.2	Atomic Processes, Systems and Runs . . . . .	7
2.3	Atomic Dolev-Yao Processes . . . . .	7
<b>3</b>	<b>Our Web Model</b>	<b>8</b>
3.1	Web System . . . . .	8
3.2	DNS Servers . . . . .	10
3.3	HTTP Messages . . . . .	10
3.4	Web Browsers . . . . .	11
3.4.1	Browser State: $Z^p$ and $s_0^p$ . . . . .	11
3.4.2	Web Browser Relation $R^p$ . . . . .	13
3.5	Web Servers . . . . .	17
3.6	Limitations . . . . .	18
<b>4</b>	<b>The BrowserID System</b>	<b>18</b>
4.1	Overview . . . . .	18
4.2	Implementation Details . . . . .	19
4.3	Sideshow and BigTent . . . . .	22
4.4	Secondary Identity Provider . . . . .	23
<b>5</b>	<b>Analysis of BrowserID</b>	<b>23</b>
5.1	Security Properties for BrowserID . . . . .	23
5.2	Attacks on BrowserID . . . . .	24
5.2.1	Identity Forgery . . . . .	24
5.2.2	Login Injection Attack . . . . .	24
5.2.3	Key Cleanup Failure Attack . . . . .	25
5.2.4	Cookie Cleanup Failure Attack (for the case of secondary IdP only) . . . . .	25
5.3	Analysis of BrowserID with sIdP . . . . .	25
5.4	Security of the Fixed System . . . . .	27
<b>6</b>	<b>Related Work</b>	<b>27</b>
<b>7</b>	<b>Conclusion</b>	<b>28</b>
<b>A</b>	<b>Message and Data Formats</b>	<b>28</b>
A.1	Notations . . . . .	29
A.2	URLs . . . . .	30
A.3	Origins . . . . .	30
A.4	Cookies . . . . .	30
A.5	HTTP Messages . . . . .	30
A.6	DNS Messages . . . . .	31

<b>B Detailed Description of the Browser Model</b>	<b>32</b>
B.1 Notation and Terminology (Web Browser State) . . . . .	32
B.2 Description of the Web Browser Atomic Process . . . . .	34
B.2.1 Functions . . . . .	34
B.2.2 Main Algorithm . . . . .	40
<b>C DNS Servers</b>	<b>41</b>
<b>D Step-By-Step Description of BrowserID (Primary IdP)</b>	<b>42</b>
<b>E Sideshow/BigTent OpenID Flow</b>	<b>44</b>
E.1 OpenID Authentication Request . . . . .	44
E.2 OpenID Authentication Response . . . . .	44
E.3 Verification . . . . .	45
<b>F Step-By-Step Description of BrowserID (Secondary IdP)</b>	<b>47</b>
F.1 Additional Checks . . . . .	48
F.2 Automatic CAP Creation . . . . .	49
F.3 LPO Session . . . . .	49
F.4 Logout . . . . .	49
<b>G BrowserID Model</b>	<b>49</b>
G.1 Attacker . . . . .	50
G.2 Browsers . . . . .	50
G.3 LPO . . . . .	50
G.4 Relying Parties . . . . .	54
G.5 BrowserID Scripts . . . . .	55
G.5.1 Some Notation and Helper Functions . . . . .	55
G.5.2 <i>script_LPO_cif</i> . . . . .	57
G.5.3 <i>script_LPO_ld</i> . . . . .	60
G.5.4 <i>script_RP_index</i> . . . . .	63
<b>H Security Property</b>	<b>66</b>
<b>I Proof of Theorem 1</b>	<b>66</b>
I.1 Condition A . . . . .	66
I.1.1 The attacker does not know $k$ in $s_j$ . . . . .	74
I.1.2 The attacker knows $k$ in $s_j$ . . . . .	75
I.2 Condition B . . . . .	76

## 1 Introduction

The World Wide Web is a complex infrastructure, with a rich set of security requirements and entities, such as DNS servers, web servers, and web browsers, interacting using diverse technologies. New technologies and standards (for example, HTML5 and related technologies) introduce even more complexity and security issues. As illustrated by numerous attacks (see, e.g., [2,6,21,28,31]), rigorous analysis of the web infrastructure and web applications is indispensable.

Inspired by successful prior work, in particular the work by Akhawe et al. [2] and Bansal et al. [5,6], one goal of our work is to develop an expressive formal model that precisely captures core security aspects of the web infrastructure, where we intend to stay as close to the standards as possible, with a level of abstraction that is suitable for precise formal analysis. As further discussed in Section 6, while prior work aimed at automatic analysis, here our main focus is to obtain a comprehensive and more accurate model with respect to the standards and specifications. As such, our model constitutes a solid basis for the analysis of a broad range of standards and applications.

The standards and specifications that define the web are spread across many documents, including the HTTP standard RFC2616 (with its successor HTTPbis) and the HTML5 specification [18], with certain aspects covered in related documents, such as RFC6265, RFC6797, RFC6454, the WHATWG Fetch living standard [33], the W3C Web Storage specification [32], and the W3C Cross-Origin Resource Sharing specification [12], to name just a few. Specifications for the DNS system and communication protocols, such as TCP, are relevant as well. The documents often build upon each other, replace older versions or other documents, and sometimes different versions coexist. Some details or behaviors are not specified at all and are only documented in the form of the source code of a web browser.

Coming up with an accurate formal model is, hence, very valuable not only because it is required as a basis to precisely state security properties and perform formal analysis, but also because it summarizes and condenses important aspects in several specifications that are otherwise spread across different documents.

Another goal and important contribution of our work is to apply our model to the BrowserID system (also known under the marketing name *Mozilla Persona*), a complex real-world single sign-on system developed by Mozilla. BrowserID makes heavy use of several web technologies, including AJAX, cross-document messaging (postMessages), and HTML5 web storage, and as such, is a very suitable and practically relevant target to demonstrate the importance of a comprehensive and accurate model.

More precisely, the main contributions of our work can be summarized as follows.

**Web model.** We propose a formal model of the web infrastructure and web applications. Our model is based on a general Dolev-Yao-style communication model, in which processes have addresses (modeling IP addresses) and, as usual in Dolev-Yao-style models for cryptographic protocols (see, e.g., [1]), messages are modeled as formal terms, with properties of cryptographic primitives, such as encryption and digital signatures, expressed as equational theories on terms.

As mentioned before, our model is intended to be expressive and close to the standards and specifications, while providing a suitable level of abstraction. Our model includes web servers, web browsers, and DNS servers. We model HTTP(S) requests and responses, including several headers, such as host, cookie, location, strict-transport-security (STS), and origin headers. Our model of web browsers captures the concepts of windows, documents, and iframes as well as new technologies, such as web storage and cross-document messaging. It takes into account

the complex security restrictions that are applied when accessing or navigating other windows. JavaScript is modeled in an abstract way by what we call scripting processes. These processes can be sent around and, among others, they can create iframes and initiate XMLHttpRequests (XHRs). We also consider two ways of dynamically corrupting browsers. Altogether, our model is the most comprehensive model for the web infrastructure to date (see also Section 6).

**Analysis of the BrowserID system.** We use our model to perform the first rigorous security analysis of the BrowserID system, which supports both so-called primary and secondary identity providers. Our security analysis reveals a number of very critical and previously unknown flaws, most of which cannot be captured by previous models (see Section 6). The most severe attack allows an adversary to login to any service that supports authentication via BrowserID with the email address of *any* Gmail and Yahoo user (without knowing the Gmail/Yahoo credentials of these users), hence, breaking the system completely. Another critical attack allows an attacker to force a user to login with the attacker’s identity. We confirmed that the attacks work on the actual BrowserID implementation. We propose fixes and formulate relevant security properties. For the BrowserID system with a secondary identity provider, we prove that the fixed system satisfies these properties in our model. By this, we provide the first rigorous formal analysis of the BrowserID system. Our attacks have been acknowledged by Mozilla, with the fixes for the most severe problems having been adopted by Mozilla already and other fixes being under discussion. Our findings have been rewarded by the Mozilla Security Bug Bounty Program.

**Structure of this Paper.** In Section 2, we present the basic communication model. Our web model is introduced in Section 3. For our case study, we first, in Section 4, provide a description of the BrowserID system. We then, in Section 5, present the analysis of BrowserID using our model. Related work is discussed in Section 6. We conclude in Section 7. Full details are provided in the appendix, including some notational conventions.

## 2 Communication Model

We now present a generic Dolev-Yao-style communication model on which our web model (see Section 3) is based. While the model is stated in a concise mathematical fashion, instantiations, for example, using the applied pi-calculus [1] or multi-set rewriting [13], are conceivable.

The main entities in the communication model are what we call *atomic processes*, which in Section 3 are used to model web browsers, web servers, DNS servers as well as web and network attackers. Each atomic process has a list of addresses (representing IP addresses) it listens to. A set of atomic processes forms what we call a *system*. The different atomic processes in such a system can communicate via events, which consist of a message as well as a receiver and a sender address. In every step of a run one event is chosen non-deterministically from the current “pool” of events and is delivered to an atomic process that listens to the receiver address of that event; if different atomic processes can listen to the same address, the atomic process to which the event is delivered is chosen non-deterministically among the possible processes. The (chosen) atomic process can then process the event and output new events, which are added to the pool of events, and so on. (In our web model, presented in Section 3, only network attackers may listen to addresses of other atomic processes.)

$$\begin{aligned} \text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) &= x & (1) \\ \text{dec}_s(\text{enc}_s(x, y), y) &= x & (2) \\ \text{extractmsg}(\text{sig}(x, y)) &= x & (3) \\ \text{checksig}(\text{sig}(x, y), \text{pub}(y)) &= \top & (4) \\ \pi_i(\langle x_1, \dots, x_n \rangle) &= x_i \text{ if } 1 \leq i \leq n & (5) \\ \pi_j(\langle x_1, \dots, x_n \rangle) &= \diamond \text{ if } j \notin \{1, \dots, n\} & (6) \\ \pi_j(t) &= \diamond \text{ if } t \text{ is not a sequence} & (7) \end{aligned}$$

**Figure 1.** Equational theory for  $\Sigma$ .

## 2.1 Terms, Messages and Events

To define the communication model just sketched, we first define, as usual in Dolev-Yao models, messages, such as HTTP messages, as formal terms over a signature, and based on this notion of messages, we introduce events.

The signature  $\Sigma$  for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

- constants  $C = \text{Addresses} \cup \mathbb{S} \cup \{\top, \perp, \diamond\}$  where the three sets are pairwise disjoint,  $\mathbb{S}$  is interpreted to be the set of ASCII strings (including the empty string  $\varepsilon$ ), and **Addresses** is interpreted to be a set of (IP) addresses,
- function symbols for public keys, asymmetric/symmetric encryption/decryption, and signatures:  $\text{pub}(\cdot)$ ,  $\text{enc}_a(\cdot, \cdot)$ ,  $\text{dec}_a(\cdot, \cdot)$ ,  $\text{enc}_s(\cdot, \cdot)$ ,  $\text{dec}_s(\cdot, \cdot)$ ,  $\text{sig}(\cdot, \cdot)$ ,  $\text{checksig}(\cdot, \cdot)$ ,  $\text{extractmsg}(\cdot)$ ,
- $n$ -ary sequences  $\langle \cdot \rangle$ ,  $\langle \cdot, \cdot \rangle$ ,  $\langle \cdot, \cdot, \cdot \rangle$ , etc., and
- projection symbols  $\pi_i(\cdot)$  for all  $i \in \mathbb{N}$ .

Let  $X = \{x_0, x_1, \dots\}$  be a set of variables and  $\mathcal{N}$  be an infinite set of constants (*nonces*) such that  $\Sigma$ ,  $X$ , and  $\mathcal{N}$  are pairwise disjoint. For  $N \subseteq \mathcal{N}$ , we define the set  $\mathcal{T}_N(X)$  of *terms* over  $\Sigma \cup N \cup X$  inductively as usual: (1) If  $t \in N \cup X$ , then  $t$  is a term. (2) If  $f \in \Sigma$  is an  $n$ -ary function symbol in  $\Sigma$  for some  $n \geq 0$  and  $t_1, \dots, t_n$  are terms, then  $f(t_1, \dots, t_n)$  is a term. By  $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$ , we denote the set of all terms over  $\Sigma \cup N$  without variables, called *ground terms*. The set  $\mathcal{M}$  of messages (over  $\mathcal{N}$ ) is defined to be the set of ground terms  $\mathcal{T}_{\mathcal{N}}$ . For example,  $k \in \mathcal{N}$  and  $\text{pub}(k)$  are messages, where  $k$  typically models a private key and  $\text{pub}(k)$  the corresponding public key. For constants  $a, b, c$  and the nonce  $k \in \mathcal{N}$ , the message  $\text{enc}_a(\langle a, b, c \rangle, \text{pub}(k))$  is interpreted to be the message  $\langle a, b, c \rangle$  (the sequence of constants  $a, b, c$ ) encrypted by the public key  $\text{pub}(k)$ .

For strings, i.e., elements in  $\mathbb{S}$ , we use a specific font. For example, **HTTPReq** and **HTTPResp** are strings. We denote by **Domains**  $\subseteq \mathbb{S}$  the set of domains, e.g., **www.example.com**  $\in$  **Domains**. We denote by **Methods**  $\subseteq \mathbb{S}$  the set of methods used in HTTP requests, e.g., **GET**, **POST**  $\in$  **Methods**.

The equational theory associated with the signature  $\Sigma$  is given in Figure 1.

By  $\equiv$  we denote the congruence relation on  $\mathcal{T}_{\mathcal{N}}(X)$  induced by this theory. For example, we have that  $\pi_1(\text{dec}_a(\text{enc}_a(\langle \mathbf{a}, \mathbf{b} \rangle, \text{pub}(k)), k)) \equiv \mathbf{a}$ .

An *event* (over **Addresses** and  $\mathcal{M}$ ) is of the form  $(a:f:m)$ , for  $a, f \in \mathbf{Addresses}$  and  $m \in \mathcal{M}$ , where  $a$  is interpreted to be the receiver address and  $f$  is the sender address. We denote by  $\mathcal{E}$  the set of all events.

## 2.2 Atomic Processes, Systems and Runs

We now define atomic processes, systems, and runs of systems.

An atomic process takes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

**Definition 1.** A (generic) atomic process is a tuple  $p = (I^p, Z^p, R^p, s_0^p)$  where  $I^p \subseteq \mathbf{Addresses}$ ,  $Z^p$  is a set of states,  $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}} \times Z^p)$ , and  $s_0^p \in Z^p$  is the initial state of  $p$ . We write  $(e, z)R(E, z')$  instead of  $((e, z), (E, z')) \in R$ .

A system  $\mathcal{P}$  is a (possibly infinite) set of atomic processes.

In order to define a run of a system, we first define configurations and processing steps.

A *configuration* of a system  $\mathcal{P}$  is a tuple  $(S, E)$  where  $S$  maps every atomic process  $p \in \mathcal{P}$  to its current state  $S(p) \in Z^p$  and  $E$  is a (possibly infinite) multi-set of events waiting to be delivered.

A *processing step* of the system  $\mathcal{P}$  is of the form  $(S, E) \xrightarrow[p \rightarrow E_{\text{out}}]{e \rightarrow p} (S', E')$  such that

- there exist  $e = (a:f:m) \in E$ ,  $E_{\text{out}} \subseteq E'$ , and  $p \in \mathcal{P}$  with  $(e, S(p))R^p(E_{\text{out}}, S'(p))$  and  $a \in I^p$ ,
- $S'(p') = S(p')$  for all  $p' \neq p$ , and
- $E' = (E \setminus \{e\}) \cup E_{\text{out}}$  (multi-set operations).

We may omit the superscript and/or subscript of the arrow.

**Definition 2.** Let  $\mathcal{P}$  be a system and  $E_0$  be a multi-set of events. A run  $\rho$  of a system  $\mathcal{P}$  initiated by  $E_0$  is a finite sequence of configurations  $(S_0, E_0), \dots, (S_n, E_n)$  or an infinite sequence of configurations  $(S_0, E_0), \dots$  such that  $S_0(p) = s_0^p$  for all  $p \in \mathcal{P}$  and  $(S_i, E_i) \rightarrow (S_{i+1}, E_{i+1})$  for all  $0 \leq i < n$  (finite run) or for all  $i \geq 0$  (infinite run).

## 2.3 Atomic Dolev-Yao Processes

We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Let  $N \subseteq \mathcal{N}$ ,  $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$ , and  $t_1, \dots, t_n \in \mathcal{T}_N$ . Then, by  $\tau[t_1/x_1, \dots, t_n/x_n]$  we denote the (ground) term obtained from  $\tau$  by replacing all occurrences of  $x_i$  in  $\tau$  by  $t_i$ , for all  $i \in \{1, \dots, n\}$ . Let  $M \subseteq \mathcal{M}$  be a set of messages. We say that a message  $m$  can be derived from  $M$  with nonces  $N$  if there exist  $n \geq 0$ ,  $m_1, \dots, m_n \in M$ , and  $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$  such that  $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$ . We denote by  $d_N(M)$  the set of all messages that can be derived from  $M$  with nonces  $N$ . For example,  $a \in d_{\{k\}}(\{\text{enc}_a(\langle a, b, c \rangle), \text{pub}(k)\})$ .

**Definition 3.** An atomic Dolev-Yao process (or simply, a DY process) is a tuple  $p = (I^p, Z^p, R^p, s_0^p, N^p)$  such that  $(I^p, Z^p, R^p, s_0^p)$  is an atomic process and (1)  $N^p \subseteq \mathcal{N}$  is an (initial) set of nonces, (2)  $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$  (and hence,  $s_0^p \in \mathcal{T}_{\mathcal{N}}$ ), and (3) for all  $a, a', f, f' \in \mathbf{Addresses}$ ,  $m, m', s, s' \in$

$\mathcal{T}_{\mathcal{N}}$ , set of events  $E$  with  $((a:f:m), s)R(E, s')$  and  $(a':f':m') \in E$  it holds true that  $m', s' \in d_N(\{m, s\})$ . (Note that  $a', f' \in d_N(\{m, s\})$ .)

In the rest of this paper, we will only consider DY processes and assume different DY processes to have disjoint initial sets of nonces.

We define a specific DY process, called an attacker process, which records all messages it receives and outputs all messages it can possibly derive from its recorded messages. Hence, an attacker process is the maximally powerful DY process. It can carry out all attacks any DY process could possibly perform. The attacker process is parametrized by the set of sender addresses it may use.

**Definition 4.** An (atomic) attacker process for a set of sender addresses  $A \subseteq \text{Addresses}$  is an atomic DY process  $p = (I, Z, R, s_0, N)$  such that for all  $a, f \in \text{Addresses}$ ,  $m \in \mathcal{T}_{\mathcal{N}}$ , and  $s \in Z$  we have that  $((a:f:m), s)R(E, s')$  iff  $s' = \langle \langle a, f, m \rangle, s \rangle$  and  $E = \{(a':f':m') \mid a' \in \text{Addresses}, f' \in A, m' \in d_N(\{m, s\})\}$ .

### 3 Our Web Model

We now present our web model. We formalize the web infrastructure and web applications by what we call a web system. A web system, among others, contains a (possibly infinite) set of DY processes, which model web browsers, web servers, DNS servers as well as web and network attackers.

As already mentioned in the introduction, the model has been carefully designed, closely following published (de-facto) standards, for instance, the HTTP/1.1 standard, associated (proposed) standards (mainly RFCs), and the HTML5 W3C candidate recommendation. We also checked these standards against the actual implementations (primarily, Chromium and Firefox).

#### 3.1 Web System

Before we can define a web system, we define scripting processes, which model client-side scripting technologies, such as JavaScript, in our browser model. Scripting processes are defined similarly to DY processes.

**Definition 5.** A scripting process (or simply, a script) is a relation  $R \subseteq (\mathcal{T}_{\mathcal{N}} \times 2^{\mathcal{N}}) \times \mathcal{T}_{\mathcal{N}}$  such that for all  $s, s' \in \mathcal{T}_{\mathcal{N}}$  and  $N \subseteq \mathcal{N}$  with  $(s, N)R s'$  it follows that  $s' \in d_N(s)$ .

A script is called by the browser which provides it with a (fresh, infinite) set  $N$  of nonces and state information  $s$ . The script then outputs a term  $s'$ , which represents the new internal state and some command which is interpreted by the browser (see Section 3.4 for details).

Similarly to an attacker process, we define the *attacker script*  $R^{\text{att}}$ . This script outputs everything that is derivable from the input, i.e.,  $R^{\text{att}} = \{((s, N), s') \mid s \in \mathcal{T}_{\mathcal{N}}, N \subseteq \mathcal{N}, s' \in d_N(s)\}$ .

We can now define web systems, where we distinguish between web and network attackers. Unlike web attackers, network attackers can listen to addresses of other parties and can spoof the sender address, i.e., they can control the network. Typically, a web system has either one network attacker or one or more web attackers, as network attackers subsume all web attackers. As we will see later, web and network attacks may corrupt other entities, such as browsers.

**Definition 6.** A web system  $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E_0)$  is a tuple with its components defined as follows:

The first component,  $\mathcal{W}$ , denotes a system (a set of DY processes) and is partitioned into the sets **Hon**, **Web**, and **Net** of honest, web attacker, and network attacker processes, respectively. We require that all DY processes in  $\mathcal{W}$  have disjoint sets of nonces, i.e.,  $N^p \cap N^{p'} = \emptyset$  for every distinct  $p, p' \in \mathcal{W}$ .

Every  $p \in \text{Web} \cup \text{Net}$  is an attacker process for some set of sender addresses  $A \subseteq \text{Addresses}$ . For a web attacker  $p \in \text{Web}$ , we require its set of addresses  $I^p$  to be disjoint from the set of addresses of all other web attackers and honest processes, i.e.,  $I^p \cap I^{p'} = \emptyset$  for all  $p' \in \text{Hon} \cup \text{Web}$ . Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that  $A = I^p$ , i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on  $I^p$ ) and may spoof all addresses (i.e., the set  $A$  may be **Addresses**).

Every  $p \in \text{Hon}$  is a DY process which models either a web server, a web browser, or a DNS server, as further described in the following subsections. Just as for web attackers, we require that  $p$  does not spoof sender addresses and that its set of addresses  $I^p$  is disjoint from those of other honest processes and the web attackers.

The second component,  $\mathcal{S}$ , is a finite set of scripts such that  $R^{\text{att}} \in \mathcal{S}$ . The third component, **script**, is an injective mapping from  $\mathcal{S}$  to  $\mathbb{S}$ , i.e., by **script** every  $s \in \mathcal{S}$  is assigned its string representation **script**( $s$ ).

Finally,  $E_0$  is a multi-set of events, containing an infinite number of events of the form ( $a:a:\text{TRIGGER}$ ) for every  $a \in \bigcup_{p \in \mathcal{W}} I^p$ .

A run of  $\mathcal{WS}$  is a run of  $\mathcal{W}$  initiated by  $E_0$ .

In the definition above, the multi-set  $E_0$  of initial events contains for every process and address an infinite number of **TRIGGER** messages in order to make sure that every process in  $\mathcal{W}$  can be triggered arbitrarily often. In particular, by this it is guaranteed that an adversary (a dishonest server/browser) can be triggered arbitrarily often. Also, we use trigger events to model that an honest browser takes an action triggered by a user, who might, for example, enter a URL or click on some link.

The set  $\mathcal{S} \setminus \{R^{\text{att}}\}$  specified in a web system as defined above is meant to describe the set of honest scripts used in the considered web application. These scripts are those sent out by an honest web server to a browser as part of a web application. In real web applications, possibly several dynamically loaded scripts may run in one document. However, if these scripts originate from honest sites, their composition can be considered to be one honest script (which is loaded right from the start into the document). In this sense, every script in  $\mathcal{S} \setminus \{R^{\text{att}}\}$  models an honest script or a combination of such scripts in a web application. (In our case study, the combination is illustrated by the script running in **RP-Doc**.)

We model the situation where some malicious script was loaded into a document by the “worst-case” scenario, i.e., we allow such a script to be the script  $R^{\text{att}}$ . This script subsumes *everything* any malicious (and honest) script can do.

We emphasize that script representations being modeled as strings are public information, i.e., any server or attacker is free to send out the string representation for any script.

Since we do not model client-side or server-side language details, and hence details such as correct escaping of user input, we cannot analyze whether a server application (say, written in PHP) is vulnerable to Cross-Site-Scripting. However, we can model the effects of Cross-Site-Scripting by letting the (model of the) server output the script  $R^{\text{att}}$ , say, if it receives certain

malicious input.

In the following subsections, (honest) DNS servers and web browsers are modeled as DY processes, including the modeling of HTTP messages. We also discuss the modeling of web servers.

### 3.2 DNS Servers

For the sake of brevity, in this paper we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover certain attacks on the DNS system itself.

A *DNS server*  $d$  (in a flat DNS model) is modeled in a straightforward way as a DY process  $(I^d, \{s_0^d\}, R^d, s_0^d, N^d)$ . It has a finite set of addresses  $I^d$  and its initial (and only) state  $s_0^d$  encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle \langle \text{domain}_1, a_1 \rangle, \langle \text{domain}_2, a_2 \rangle, \dots \rangle .$$

DNS queries are answered according to this table. DNS queries have the following form, illustrated by an example:

$$\langle \text{DNSResolve}, \text{example.com}, n \rangle$$

where `example.com` is the domain name to be resolved and  $n$  is a nonce representing the random query ID and UDP source port number selected by the sender of the query. The corresponding response is of the form

$$\langle \text{DNSResolved}, a, n \rangle$$

where  $a \in \text{Addresses}$  is the IP address of the queried domain name and  $n$  is the nonce from the query.

The precise message format of DNS messages is provided in Appendix A.6, the full formal specification for DNS servers can be found in Appendix C.

### 3.3 HTTP Messages

In order to model web browsers and servers, we first need to model HTTP requests and responses. The formal specification of HTTP messages can be found in Appendix A.5. Here we provide a more informal presentation.

HTTP requests and responses are encoded as messages (ground terms). An HTTP request (modeled as a message) contains a nonce, a method (e.g., `GET` or `POST`), a domain name, a path, URL parameters, request headers (such as `Cookie` or `Origin`), and a message body. For example, an HTTP `GET` request for the URL `http://example.com/show?page=1` is modeled as the term

$$r := \langle \text{HTTPReq}, n_1, \text{GET}, \text{example.com}, /show, \langle \langle \text{page}, 1 \rangle \rangle, \langle \rangle, \langle \rangle \rangle$$

where body and headers are empty. A web server that responds to this request is supposed to include the nonce  $n_1$  contained in  $r$  in the response so that the browser can match the request to the corresponding response. More specifically, an HTTP response (modeled as a

message) contains a nonce (matching the request), a status code (e.g., 200 for a normal successful response), response headers (such as `Set-Cookie` and `Location`), and a body. For example, a response to  $r$  could be

$$s := \langle \text{HTTPResp}, n_1, 200, \langle \langle \text{Set-Cookie}, \langle \text{SID}, \langle n_2, \perp, \top, \perp \rangle \rangle \rangle, \langle \text{script1}, n_3 \rangle \rangle$$

where  $s$  contains (1) in the headers section, a cookie with the name `SID`, the value  $n_2$ , and the attributes `secure` and `httpOnly` not set but the attribute `session` set (see Section 3.4 for details on cookies) and (2) in the body section, the string representation `script1` of the scripting process  $\text{script}^{-1}(\text{script1})$  (which should be an element of  $\mathcal{S}$ ) and its initial state  $n_3$ .

For the HTTP request and response in the above examples, the corresponding HTTPS request would be of the form  $\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}}))$  and the response of the form  $\text{enc}_s(s, k')$  where  $k'$  is a fresh symmetric key (a nonce) which is typically generated by the sender of the request. The responder is supposed to use this key to encrypt the response.

### 3.4 Web Browsers

We think of an honest browser to be used by one honest user. However, we also allow browsers to be taken over by attackers. The honest user is modeled as part of the web browser model. Actions a user takes are modeled as non-deterministic actions of the web browser. For example, the web browser itself can non-deterministically follow the links provided by a web page. Secrets, such as passwords, typically provided by the user are stored in the initial state of a browser and are given to a web page when needed, similar to the `AutoFill` function in browsers (see below).

A web browser  $p$  is modeled as a DY process  $(I^p, Z^p, R^p, s_0^p, N^p)$  where  $I^p \subseteq \text{Addresses}$  is a finite set and  $N^p \subseteq \mathcal{X}$  is an infinite set. The set of states  $Z^p$ , the initial state  $s_0^p$ , and the relation  $R^p$  are defined below (Sections 3.4.1 and 3.4.2), with a full formal specification provided in Appendix B.

#### 3.4.1 Browser State: $Z^p$ and $s_0^p$

The set  $Z^p$  of states of a browser consists of terms of the form

$$\langle \text{windows}, \text{secrets}, \text{cookies}, \text{localStorage}, \text{sessionStorage}, \text{keyMapping}, \text{sts}, \text{DNSaddress}, \text{nonces}, \text{pendingDNS}, \text{pendingRequests}, \text{isCorrupted} \rangle$$

**Windows and documents.** The most important part of the state are windows and documents, both stored in the subterm *windows*. A browser may have a number of windows open at any time (resembling the tabs in a real browser). Each window contains a list of documents of which one is “active”. Being active means that this document is currently presented to the user and is available for interaction, similarly to the definition of active documents in the HTML5 specification [18]. The document list of a window represents the history of visited web pages in that window. A window may be navigated forward and backward (modeling forward and back buttons). This deactivates one document and activates its successor or predecessor.

A document is specified by a term which essentially contains (the string representing) a script, the current state of the script, the input that the script obtained so far (from XHRs and `postMessages`), the origin (domain name plus HTTP or HTTPS, see Appendix A.3 for details) of the document, and a list of windows (called *subwindows*), which correspond to iframes embedded in the document, resulting in a tree of windows and documents. The (single) script is meant

to model the static HTML code, including, for example, links and forms, and possibly multiple JavaScript code parts. When called by the browser, a script essentially outputs a command which is then interpreted by the browser, such as following a link, creating an iframe, or issuing an XHR. In particular, a script can represent a plain HTML document consisting merely of links, say: when called by the browser such a script would non-deterministically choose such a link and output it to the browser, which would then load the corresponding web page (see below for details).

We use the terms *top-level window* (a window which is not a subwindow itself), *parent window* (the window of which the current window is a direct subwindow) and *ancestor window* (some window of which the current window is a not necessarily direct subwindow) to describe the relationships in a tree of windows and documents.

A term describing a window or a document also contains a unique nonce, which we refer to by *reference*. This reference is used to match HTTP responses to the corresponding windows and documents from which they originate (see below).

Top-level windows may have been opened by another window. In this case, the term of the opened window contains a reference to the window by which it was opened (the *opener*). Following the HTML5 standard, we call such a window an *auxiliary window*. Note that auxiliary windows are always top-level windows.

We call a window *active* if it is a top-level window or if it is a subwindow of an active document in an active window. Note that the active documents in all active windows are exactly those documents a user can currently see/interact with in the browser.

**Example 1.** *The following is an example of a window term with reference  $n_1$ , two documents, and an opener ( $n_4$ ):*

$$\langle n_1, \langle \langle n_2, \langle \text{example.com}, \text{P} \rangle, \text{script1}, \langle \rangle, \langle \rangle, \langle \rangle, \perp \rangle, \langle n_3, \langle \text{example.com}, \text{S} \rangle, \text{script2}, \langle \rangle, \langle \rangle, \langle \rangle, \top \rangle \rangle, n_4 \rangle$$

*The first document has reference  $n_2$ . It was loaded from the origin  $\langle \text{example.com}, \text{P} \rangle$ , which translates into <http://example.com>. Its scripting process has the string representation `script1`, the last state and the input history of this process are empty. The document does not have subwindows and is inactive ( $\perp$ ). The second document has the reference  $n_3$ , its origin corresponds to <https://example.com>, the scripting process is represented by `script2`, and the document is active ( $\top$ ). All other components are empty.*

**Secrets.** This subterm of the state term of a browser holds the secrets of the user of the web browser. Secrets (such as passwords) are modeled as nonces and they are indexed by origins. Secrets are only released to documents (scripts) with the corresponding origin, similarly to the AutoFill mechanism in browsers.

**Cookies, localStorage, and sessionStorage.** These subterms contain the cookies (indexed by domains), localStorage data (indexed by origins), and sessionStorage data (indexed by origins and top-level window references) stored in the browser. Cookies are stored together with their `secure`, `httpOnly`, and `session` attributes: If `secure` is set, the cookie is only delivered to HTTPS origins. If `httpOnly` is set, the cookie cannot be accessed by JavaScript (the script). According to the proposed standard RFC6265 (which we follow in our model) and the majority of the existing implementations, cookies that neither have the (real) “max-age” nor the “expires” attribute should be deleted by the browser when the session ends (usually when the browser is closed). In our model, such cookies carry the `session` attribute.

**KeyMapping.** This term is our equivalent to a certificate authority (CA) certificate store in the browser. Since, for simplicity, we currently do not formalize CAs in the model, this term simply encodes a mapping assigning domains  $d \in \text{Domains}$  to their respective public keys  $\text{pub}(k_d)$ .

**STS.** Domains that are listed in this term are contacted by the web browser only over HTTPS. Connection attempts over HTTP are transparently rewritten to HTTPS requests. Web sites can issue the **Strict-Transport-Security** header to clients in order to add their domain to this list, see below.

**DNSaddress.** This term contains the address of the DY process that is to be contacted for DNS requests; typically a DNS server.

**Nonces, pendingDNS, and pendingRequests.** These terms are used for bookkeeping purposes, recording the nonces that have been used by the browser so far, the HTTP requests that await successful DNS resolution, and HTTP requests that await a response, respectively.

**IsCorrupted.** This term indicates whether the browser is corrupted ( $\neq \perp$ ) or not ( $= \perp$ ). A corrupted browser behaves like a web attacker (see Section 3.4.2).

**Initial state  $s_0^p$  of a web browser.** In the initial state, *keyMapping*, *DNSAddress*, and *secrets* are defined as needed, *isCorrupted* is set to  $\perp$ , and all other subterms are  $\langle \rangle$ .

### 3.4.2 Web Browser Relation $R^p$

Before we define the relation  $R^p$ , we first sketch the processing of HTTP(S) requests and responses by a web browser, and also provide some intuition about the corruption of browsers.

**HTTP(S) Requests and Responses.** An HTTP request, contains, as mentioned before, a nonce created by the browser. In the example in Section 3.3, this nonce is  $n_1$ . A server is supposed to include this nonce into its HTTP response. By this, the browser can match the response to the request (a real web browser would use the TCP sequence number for this purpose). If a browser wants to send an HTTP request, it first resolves the domain name to an IP address. (For simplicity, we do not model DNS response caching.) It therefore first records the HTTP request in *pendingDNS* along with the reference of the window (in the case of HTTP(S) requests) or the reference of the document<sup>1</sup> (in the case of XHRs) from which the request originated and then sends a DNS request. Upon receipt of the corresponding DNS response it sends the HTTP request and stores it (again along with the reference as well as the server address) in *pendingRequests*. Before sending the HTTP request, the cookies stored in the browser for the domain of the request are added as cookie headers to the request. Cookies with attribute **secure** are only added for HTTPS requests. If an HTTP response arrives, the browser uses the nonce in this response to match it with the recorded corresponding HTTP request (if any) and checks whether the address of the sender is as expected. The reference recorded along with the request then determines to which window/document the response belongs. The further processing of a response is described below.

We note that before HTTPS requests are sent out, a fresh symmetric key (a nonce) is generated and added to the request by the browser. The resulting message is then encrypted using the public key corresponding to the domain in the request (according to *keyMapping*). The symmetric key is recorded along with the request in *pendingRequests*. The response is,

<sup>1</sup>As we will see later, in the case of XHRs this reference is actually a sequence of two elements, a document reference and a nonce that was chosen by the script that issued the XHR. For now, we will refer to this sequence simply as the document reference.

PROCESSING INPUT MESSAGE  $m$  $m = \text{FULLCORRUPT}$ :  $isCorrupted := \text{FULLCORRUPT}$  $m = \text{CLOSECORRUPT}$ :  $isCorrupted := \text{CLOSECORRUPT}$  $m = \text{TRIGGER}$ : non-deterministically choose  $action$  from  $\{1, 2\}$  $action = 1$ : Call script of some active document. Outputs new state and command  $cmd$ . $cmd = \text{HREF}$ :  $\rightarrow$  *Initiate request* $cmd = \text{IFRAME}$ : Create subwindow,  $\rightarrow$  *Initiate request* $cmd = \text{FORM}$ :  $\rightarrow$  *Initiate request* $cmd = \text{SETSCRIPT}$ : Change script in given document. $cmd = \text{SETSCRIPTSTATE}$ : Change state of script in given document. $cmd = \text{XMLHTTPREQUEST}$ :  $\rightarrow$  *Initiate request* $cmd = \text{BACK or FORWARD}$ : Navigate given window. $cmd = \text{CLOSE}$ : Close given window. $cmd = \text{POSTMESSAGE}$ : Send `postMessage` to specified document. $action = 2$ :  $\rightarrow$  *Initiate request to some URL in new window* $m = \text{DNS response}$ : send corresponding HTTP request $m = \text{HTTP(S) response}$ : (decrypt,) find reference.**reference to window**: create document in window**reference to document**: add response body to document's script input

**Figure 2.** The basic structure of the web browser relation  $R^p$  with an extract of the most important processing steps, in the case that  $isCorrupted = \perp$ .

as mentioned, supposed to be encrypted with this symmetric key (see also Appendix A.5 for details).

**Corruption of Browsers.** We model two types of corruption of browsers, namely *full corruption* and *close-corruption*, which are triggered by special network messages in our model. In the real world, an attacker can exploit buffer overflows in web browsers, compromise operating systems (e.g., using trojan horses), and physically take control over shared terminals.

Full corruption models an attacker that gained full control over a web browser and its user. Besides modeling a compromised system, full corruption can also serve as a vehicle for the attacker to participate in a protocol using secrets of honest browsers: In our case study (Section 5), the attacker starts with no user secrets in its knowledge, but may fully corrupt any number of browsers, so, in particular, he is able to impersonate browsers/users.

Close-corruption models a browser that is taken over by the attacker after a user finished her browsing session, i.e., after closing all windows of the browser. This form of corruption is relevant in situations where one browser can be used by many people, e.g., in an Internet café. Information left in the browser state after closing the browser could be misused by malicious users.

**The Relation  $R^p$ .** To define  $R^p$ , we need to specify, given the current state of the browser and an input message  $m$ , the new state of the browser and the set of events output by the browser. Figure 2 provides an overview of the structure of the following definition of  $R^p$ . The input message  $m$  is expected to be FULLCORRUPT, CLOSECORRUPT, TRIGGER, a DNS response, or an HTTP(S) response.

If  $isCorrupted \neq \perp$  (browser is corrupted), the browser, just like an attacker process, simply adds  $m$  to its current state, and then outputs all events it can derive from its state. Once corrupted, the browser stays corrupted. Otherwise, if  $isCorrupted = \perp$ , on input  $m$  the browser behaves as follows.

**$m = \text{FULLCORRUPT}$ :** If the browser receives this message, it sets *isCorrupted* to **FULLCORRUPT**. From then on the browser is corrupted as described above, with the attacker having full access to the browser’s internal state, including all secrets.

**$m = \text{CLOSECORRUPT}$ :** If the browser receives this message, it first removes the user secrets, open windows and documents, all *session* cookies, all *sessionStorage* data, and all pending requests from its current state; nonces used so far by the browser may not be used any longer. *LocalStorage* data and persistent cookies are not deleted. The browser then sets *isCorrupted* to **CLOSECORRUPT** (and hence, from then on is corrupted). As already mentioned, this models that the browser is closed by a user and that then the browser is used by another, potentially malicious user (an attacker), such as in an Internet café.

**$m = \text{TRIGGER}$ :** Upon receipt of this message, the browser non-deterministically chooses one of two *actions*: (1) trigger a script or (2) request a new document.

**$m = \text{TRIGGER}$ , *action* = 1:** Some active window (possibly an *iframe*) is chosen non-deterministically. Then the script of the active document of that window is triggered (see below).

**$m = \text{TRIGGER}$ , *action* = 2:** A new HTTP(S) **GET** request (i.e., an HTTP(S) request with method **GET**) is created where the URL is some message derivable from the current state of the browser. However, nonces may not be used. This models the user typing in a URL herself, but we do not allow her to type in secrets, e.g., passwords or session tokens. A new window is created to show the response. (HTTP requests to domains listed in *sts* are automatically rewritten to HTTPS requests).

**$m = \text{DNS response}$ :** DNS responses are processed as already described above, resulting in sending the corresponding HTTP(S) request (if any).

**$m = \text{HTTP(S) response}$ :** The browser performs the steps (I) to (IV) in this order.

**(I)** The browser identifies the corresponding HTTP(S) request (if any), say  $q$ , and the window or document from which  $q$  originated. (In case of HTTPS, the browser also decrypts  $m$  using the recorded symmetric key.)

**(II)** If there is a **Set-Cookie** header in the response, its content (name, value, and if present, the attributes **httpOnly**, **secure**, **session**) is evaluated: The cookie’s name, value, and attributes are saved in the browser’s list of cookies. If a cookie with the same name already exists, the old values and attributes are overwritten, as specified in RFC6265.

**(III)** If there is a **Strict-Transport-Security** header in the response, the domain of  $q$  is added to the term *sts*. As defined in RFC6797, all future requests to this domain, if not already HTTPS requests, are automatically altered to use HTTPS.

**(IV)** If there is a **Location** header (with some URL  $u$ ) in the response and the HTTP status code is 303 or 307, the browser performs a redirection (unless it is a non-same-origin redirect of an XHR) by issuing a new HTTP request to  $u$ , retaining the body of the original request. Rewriting POST to GET requests for 303 redirects and extending the origin header value are handled as defined in RFC2616 and in the W3C Cross-Origin Resource Sharing specification [12].

Otherwise, if no redirection is requested, the browser does the following: If the request originated from a window, a new document is created from the response body. For this, the response body is expected to be a term of the form  $\langle sp, stat \rangle$  where  $sp$  is a string such that  $\text{script}^{-1}(sp) \in \mathcal{S}$  is a script and  $stat$  is a term used as its initial state. The document is then added to the window the reference points to, it becomes the active document, and the successor of the currently active document. All previously existing successors are removed. If the request originated from a document (and hence, was the result of an XHR), the body of the response is

appended to the script input term of the document. When later the script of this document is activated, it can read and process the response.

**Triggering the Script of a Document** ( $m = \text{TRIGGER}$ ,  $action = 1$ ). First, the script of the document is called with the following input:

- document and window references of all active documents and subwindows<sup>2</sup>
- only for same-origin documents: information about the documents' origins, scripts, script states and script inputs
- the last state and the input history (i.e., previous inputs from `postMessages` and `XHRs`) of the script as recorded in the document
- cookies (names and values only) indexed with the document's domain, except for `httpOnly` cookies
- `localStorage` data for the document's origin
- `sessionStorage` data that is indexed with the document's origin and the reference of the document's top-level window
- secrets indexed with the document's origin.

In addition, the script is given an infinite set of fresh nonces from the browser's set of (unused) nonces.

Now, according to the definition of scripts, the script outputs a term. The browser expects terms of the form  $\langle state, cookies, localStorage, sessionStorage, cmd \rangle$  (and otherwise ignores the output) where *state* is an arbitrary term describing the new state of the script, *cookies* is a sequence of name/value pairs, *localStorage* and *sessionStorage* are arbitrary terms, and *cmd* is a term which is interpreted as a command which is to be processed by the browser. The old state of the script recorded in the document is replaced by the new one (*state*), the local/session storage data recorded in the browser for the document's origin (and top-level window reference) is replaced by *localStorage/sessionStorage*, and the old cookie store of the document's origin is updated using *cookies* similarly to the case of HTTP(S) responses with cookie headers, except that now no `httpOnly` cookies can be set or replaced, as defined by the HTML5 standard [18] in combination with RFC6265. For details, see Line 16 of Algorithm 6 and Definition 22 in Appendix B.

Subsequently, *cmd* (if not empty) is interpreted by the browser, as described next. We note that commands may contain parameters.

***cmd* = HREF** (parameters: URL *u*, window reference *w*): A new `GET` request to *u* is initiated. If *w* is `_BLANK`, the response to the request will be shown in a new *auxiliary* window. This new window will carry the reference to its opener, namely the reference to the window in which the script was running. Otherwise, if *w* is not `_BLANK`, the window with reference *w* is navigated

---

<sup>2</sup>Note that we overapproximate here: In real-world browsers, only a limited set of window handles are available to a script. Our approach is motivated by the fact that in some cases windows can be navigated by names (without a handle). However, as we will see, specific restrictions for navigating windows and accessing/changing their data apply.

(upon receipt of the response and only if it is active) to the given URL. Navigation is subject to several restrictions.<sup>3</sup>

**cmd** = **IFRAME** (parameter: URL  $u$ , window reference  $w$ ): Provided that the active document of  $w$  is same origin, create a new subwindow in  $w$  (with a new window reference) and initiate an HTTP GET request to  $u$  for that subwindow.

**cmd** = **FORM** (parameters: URL  $u$ , method  $md$ , data  $d$ , window reference  $w$ ): Initiate a new request for  $u$  with method  $md$ , and body data  $d$ , where, just like in the case of **HREF**,  $w$  determines in what window the response is shown. Again the same restrictions for navigating other windows as in the case of **HREF** apply. For this request an **Origin** header is set if  $md = \text{POST}$ . Its value is the origin of the document.

**cmd** = **SETSCRIPT** (parameters: window reference  $w$ , string  $s$ ): Change the string representation of the script of the active document in  $w$  to  $s \in \text{script}^{-1}(\mathcal{S})$ , provided that the active document in  $w$  is same origin.

**cmd** = **SETSCRIPTSTATE** (parameters: window reference  $w$ , term  $t$ ): Change the state of the script of the active document in  $w$  to  $t$ , provided that  $w$  is same origin.

**cmd** = **XMLHTTPREQUEST** (parameters: URL  $u$ , method  $md$ , data  $d$ , nonce  $xhrreference$ ): Initiate a request with method  $md$  and data  $d$  for  $u$ , provided that  $u$  is same origin and  $md$  is not **CONNECT**, **TRACE**, or **TRACK** (according to [34]). Instead of the window reference, the reference used for this request is  $\langle r, xhrreference \rangle$ , where  $r$  is the reference of the script's document and  $xhrreference$  is a nonce chosen by the script (for later correlation). This combined reference indicates that this request originated from an XHR and  $xhrreference$  is used by the script to assign the response to the request. The **Origin** header is set as in the case of **FORM**.

**cmd** = **BACK or FORWARD** (parameter: window reference  $w$ ): replace the active document in  $w$  by its predecessor/successor in  $w$ . Again, the same restrictions for navigating windows as in the case of **HREF** apply.

**cmd** = **CLOSE** (parameter: window reference  $w$ ): close the given window, i.e., remove it from the list of windows in which it is contained. The same restrictions for navigating windows as in the case of **HREF** apply.

**cmd** = **POSTMESSAGE** (parameter: message  $msg$ , window reference  $w$ , origin  $o$ ):  $msg$ , the origin of the sending document, and a reference to its window are appended to the input history of the active document in  $w$  if that document's origin matches  $o$  or if  $o = \perp$ .

### 3.5 Web Servers

While the modeling of DNS servers and browsers is independent of specific web applications, and hence, forms the core of the model of the web infrastructure, the modeling of a *web server* heavily depends on the specific web application under consideration. Conversely, the model of a specific web application is determined by the model of the web server. We therefore do not and cannot fix a model for web servers at this point. Such a model should be provided as part of the analysis of a specific web application, as illustrated by our case study (see Section 4 and following).

---

<sup>3</sup>We follow the rules defined in [18], Subsection 5.1.4: A window  $A$  can navigate a window  $B$  if the active documents of both are same origin, or  $B$  is an ancestor window of  $A$  and  $B$  is a top-level window, or if there is an ancestor window of  $B$  whose active document has the same origin as the active document of  $A$  (including  $A$  itself). Additionally,  $A$  may navigate  $B$  if  $B$  is an auxiliary window and  $A$  is allowed to navigate the opener of  $B$ . We follow these rules closely, as can be seen in Algorithm 2 in Appendix B.

### 3.6 Limitations

We now briefly discuss main limitations of the model. As will be illustrated by our case study, our model is formulated on a level of abstraction that is suitable to capture many security relevant features of the web, and hence, a relevant class of attacks. However, as with all models, certain attacks are out of the scope of our model. For example, as already mentioned, we currently cannot reason about language details (e.g., how two JavaScripts running in the same document interact). Also, we currently do not model user interface details, such as frames that may overlap in Clickjacking attacks. Being a Dolev-Yao-style model, our model clearly does not aim at lower-level cryptographic attacks. Also, byte-level attacks, such as buffer overflows, are out of scope.

## 4 The BrowserID System

BrowserID [24] is a new decentralized single sign-on (SSO) system developed by Mozilla for user authentication on web sites. It is a complex full-fledged web application deployed in practice, with currently ~47k LOC (excluding code for Sideshow/BigTent, see below, and some libraries). It allows web sites to delegate user authentication to email providers, where users use their email addresses as identities. The BrowserID implementation makes use of a broad variety of browser features, such as XHRs, postMessage, local- and sessionStorage, cookies, etc.

We first, in Section 4.1, provide a high-level overview of the BrowserID system. A more detailed description of the BrowserID implementation is then given in Sections 4.2 to 4.4 with further details provided in Appendices D to F.

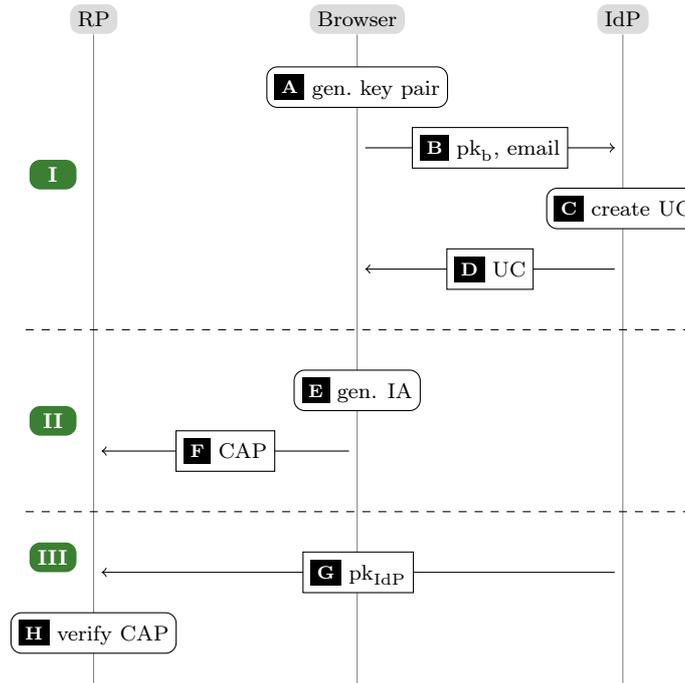
### 4.1 Overview

The BrowserID system knows three distinct parties: the user, which wants to authenticate herself using a browser, the relying party (RP) to which the user wants to authenticate (log in) with one of her email addresses (say, `user@eyedee.me`), and the identity/email address provider IdP. If the email provider (`eyedee.me`) supports BrowserID directly, it is called a *primary IdP*. Otherwise, a Mozilla-provided service, a so-called *secondary IdP*, takes the role of the IdP. In what follows, we describe the case of a primary IdP, with more information on secondary IdPs given in Section 4.4.

A primary IdP provides information about its BrowserID setup in a so-called *support document*, which it provides at a fixed URL derivable from the email domain, e.g., <https://eyedee.me/.well-known>.

A user who wants to log in at an RP with an email address for some IdP has to present two signed documents: A *user certificate* (UC) and an *identity assertion* (IA). The UC contains the user's email address and a public key. It is signed by the IdP. The IA contains the origin of the RP and is signed with the private key corresponding to the user's public key. Both documents have a limited validity period. A pair consisting of a UC and a matching IA is called a *certificate assertion pair* (CAP) or a *backed identity assertion*. Intuitively, the UC in the CAP tells the RP that (the IdP certified that) the owner of the email address is (or at least claimed to be) the owner of the public key. By the IA contained in the CAP, the RP is ensured that the owner of the given public key wants to log in. Altogether, given a valid CAP, RP would consider the user (with the email address mentioned in the CAP) to be logged in.

The BrowserID authentication process (with a primary IdP) consists of three phases (see Figure 3 for an overview): **I** provisioning of the UC, **II** CAP creation, and **III** verification of



**Figure 3.** BrowserID authentication: basic overview

the CAP.

In Phase **I**, the (browser of the) user creates a public/private key pair **A**. She then sends her public key as well as the email address she wants to use to log in at some RP to IdP **B**. IdP now creates the UC **C**, which is then sent to the user **D**. The above requires the user to be logged in at IdP.

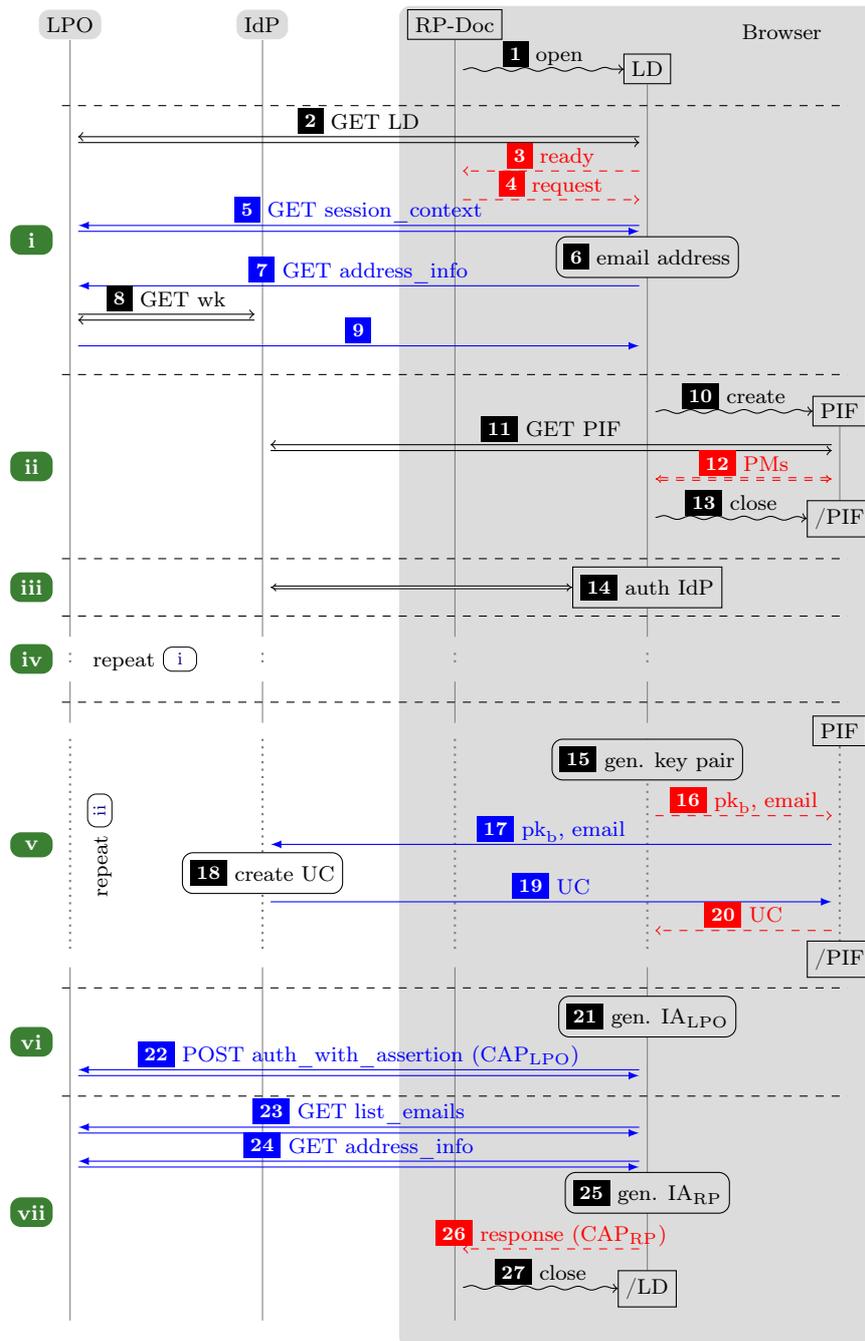
With the user having received the UC, Phase **II** can start. The user wants to authenticate to an RP, so she creates the IA **E**. The UC and the IA are concatenated to a CAP, which is then sent to the RP **F**.

In Phase **III**, the RP checks the authenticity of the CAP. For this purpose, the RP could use an external verification service provided by Mozilla or check the CAP itself as follows: First, the RP fetches the public key of IdP **G**, which is contained in the support document. Afterwards, the RP checks the signatures of the UC and the IA **H**. If this check is successful, the RP can, as mentioned before, consider the user to be logged in with the given email address and send her some token (e.g., a session ID), which we refer to as an *RP service token*.

## 4.2 Implementation Details

We now provide a more detailed description of the BrowserID implementation (see also Figure 4). Since the system is very complex, with many HTTPS requests, XHRs, and postMessages sent between different entities (servers as well as windows and iframes within the browser), we here describe mainly the phases of the login process without explaining every single message exchange done in the implementation. A more detailed step-by-step description can be found in Appendix D.

In addition to the parties mentioned in the rough overview in Section 4.1, the actual imple-



**Figure 4.** BrowserID implementation overview. Black arrows (open tips) denote HTTPS messages, blue arrows (filled tips) denote XHRs (over HTTPS), red (dashed) arrows are postMessages, snake lines are commands to the browser.

mentation uses another party, `login.persona.org` (LPO). The role of LPO is as follows: First, LPO provides the HTML and JavaScript files of the implementation. Thus, the BrowserID

implementation mainly runs under the origin of LPO.<sup>4</sup> When the JavaScript implementation running in the browser under the origin of LPO needs to retrieve information from the IdP (support document), LPO acts as a proxy to circumvent cross-origin restrictions.

Before explaining the login process, we provide a quick overview of the windows and iframes in the browser. By RP-Doc we denote the window (see Figure 4) containing the document loaded from some RP, a web page on which the user wants to log in with an email address of some IdP. This document typically includes JavaScript from LPO and contains a button “Login with BrowserID”. (Loading of *RP-Doc* from the RP and the JavaScript from LPO is not depicted in Figure 4). The LPO JavaScript running in *RP-Doc* opens an auxiliary window called the *login dialog* (LD). Its content is provided by LPO and it handles the interaction with the user. During the login process, a temporary invisible iframe called the *provisioning iframe* (PIF) can be created in the LD. The PIF is loaded from IdP. It is used by LD to communicate (cross-origin) with IdP. Temporarily, the LD may navigate itself to a web page at IdP to allow for direct user interaction with the IdP.

Now, in order to describe the login process, for the time being we assume that the user uses a “fresh” browser, i.e., the user has not been logged in before. As mentioned, the process starts by the user visiting a web site of some RP. After the user has clicked on the login button in RP-Doc, the LD is opened and the interactive login flow is started. We can divide this login flow into seven phases: In Phase (i), the LD is initialized and the user is prompted to provide her email address. Then LD fetches the support document (see Section 4.1) of IdP via LPO. In Phase (ii), LD creates the PIF from the *provisioning URL* provided in the support document. As (by our assumption) the user is not logged in yet, the PIF notifies LD that the user is not authenticated to IdP yet. In Phase (iii), LD navigates itself away to the *authentication URL* which is also provided in the support document and links to IdP. Usually, this document will show a login form in which the user enters her password to authenticate to the IdP. After the user has been authenticated to IdP (which typically implies that IdP sets a session cookie in the browser), the window is navigated to LPO again. (This is done by JavaScript loaded from LPO that the IdP document is supposed to include.)

Now, the login flow continues in Phase (iv), which basically repeats Phase (i). However, the user is not prompted for her email address (it has previously been saved in the localStorage under the origin of LPO along with a nonce, where the nonce is stored in the sessionStorage). In Phase (v), which basically repeats Phase (ii), the PIF detects that the user is now authenticated to IdP and the provisioning phase is started (I in Figure 3): The user’s keys are created by LD and stored in the localStorage under the origin of LPO. The PIF forwards the certification request to IdP, which then creates the UC and sends it back to the PIF. The PIF in turn forwards it to the LD, which stores it in the localStorage under the origin of LPO.

In Phases (vi) and (vii), mainly the IA is generated by LD for the origin of RP-Doc and sent (together with the UC) to RP-Doc (II in Figure 3). In the localStorage, LD stores that the user’s email is logged in at RP. Moreover, the user’s email is recorded at LPO (see the explanation on LPO Sessions below). For this purpose, LD generates an IA for the origin of LPO and sends the UC and IA to LPO.

**LPO Session.** LPO establishes a session with the browser by setting a cookie `browserid_state` (in Step 5 in Figure 4) on the client-side. LPO considers such a session authenticated after having received a valid CAP (in Step 22 in Figure 4). In future runs, the user is presented a list of her email addresses (which is fetched from LPO) in order to choose one address. Then, she is asked

---

<sup>4</sup>It is envisioned by Mozilla to integrate the part of LPO directly into the browser in the future.

if she trusts the computer she is using and is given the option to be logged in for one month or “for this session only” (*ephemeral* session). In order to use any of the email addresses, the user is required to authenticate to the IdP responsible for that address to get an UC issued. If the localStorage (under the origin LPO) already contains a valid UC, then, however, authentication at the IdP is not necessary.

**Automatic CAP Creation.** In addition to the interactive login presented above, BrowserID also contains an automatic, non-interactive way for RPs to obtain a freshly generated CAP: During initialization of the BrowserID code included by RP-Doc, an invisible iframe called the *communication iframe* (CIF) is created inside RP-Doc. The CIF’s JavaScript is loaded from LPO and behaves similar to LD, but without user interaction. The CIF automatically issues a fresh CAP and sends it to RP-Doc under specific conditions: among others, the email address must be marked as logged in at RP in the localStorage. If necessary, a new key pair is created and a corresponding new UC is requested at IdP.

In Figure 6 in Appendix F, we show the usage of the CIF in detail for the case of the *secondary IdP*.

**Logout.** We have to differentiate between three ways of logging out: an RP logout, an LPO logout, and an IdP logout. An RP logout is handled by the CIF after it has received a *logout* postMessage from RP-Doc. The CIF then changes the localStorage such that no email address is recorded to be logged in at RP.

An LPO logout essentially requires to logout at the web site of LPO. The LPO logout removes all key pairs and certificates from the localStorage and invalidates the session on the LPO server.

An IdP logout depends on the IdP implementation and usually cancels the user’s session with IdP. This entails that IdP will not issue new UCs for the user without re-authentication.

### 4.3 Sideshow and BigTent

Since several email providers, such as `gmail.com` and `yahoo.com`, already use OpenID [25], a widely employed SSO system, Mozilla implemented IdPs called Sideshow and BigTent which use an OpenID backend for user authentication: Sideshow/BigTent are put between BrowserID and an email provider running OpenID. That is, BrowserID uses Sideshow/BigTent as an IdP. Sideshow/BigTent translate requests from BrowserID to requests to the email provider’s OpenID interface. Currently, Sideshow and BigTent are used to provide BrowserID support for `gmail.com` and `yahoo.com`, respectively. In what follows, we describe Sideshow in more detail; BigTent is similar. Technical details on the communication between OpenID and Sideshow/BigTent can be found in Appendix E.

All BrowserID protocol steps that would normally be carried out by the IdP are now handled by Sideshow (i.e., the Sideshow server). For this purpose, Sideshow serves the provisioning URL (for the PIF) and the authentication URL used in (iii). It maintains a session with the user’s browser. This session is considered to be authenticated if the user has successfully authenticated to Sideshow using OpenID. In this case, Sideshow’s PIF document may send public keys to Sideshow. Sideshow then creates a UC for the identity it believes to be logged in. If the session at Sideshow is not authenticated, the user will first be redirected to the Sideshow authentication URL. Sideshow’s authentication document will redirect the user further to the OpenID URL at Gmail. This URL contains an authentication request encoding that Sideshow requests an *OpenID assertion* that contains an email address. In general, such an assertion is a list of attribute

name/value pairs (partially) MACed by Gmail with a temporary symmetric key known only to Gmail; an additional attribute, `openid.signed`, in such an assertion encodes which attribute name/value pairs have actually been MACed and in which order. The user now authenticates to Gmail. Then, Gmail issues the requested OpenID assertion and redirects the browser to Sideshow with the assertion in the URL parameters. Sideshow then sends the OpenID assertion to Gmail in order to check its validity. If the OpenID assertion is valid, i.e. the MAC over the attributes listed in `openid.signed` verifies, Sideshow considers its session with the user's browser to be authenticated for the email address contained in the OpenID assertion.

#### 4.4 Secondary Identity Provider

If an email provider (IdP) does not directly support BrowserID, LPO can be used as a so-called *secondary IdP* (sIdP), i.e., it replaces the IdP completely. For this, the user has to register at LPO. That is, she creates an account at LPO where she can register one or more email addresses to be used as identities. She has to prove ownership of all email addresses she registers. (LPO sends URLs to each email address, which then have to be opened by the user.)

When the sIdP is used, the phases (ii) – (vi) are not needed as now LPO replaces the IdP and the actions previously performed by IdP and LPO are now carried out by LPO alone. The user is prompted to enter her password directly into LD. If the password is correct, LPO now considers the session with the browser to be authenticated. LPO will then issue UCs on behalf of the email provider. We note that, for automatic CAP creation, the CIF (see Section 4.2) is still used.

A detailed step-by-step description of a BrowserID flow with an sIdP can be found in Appendix F.

## 5 Analysis of BrowserID

In this section, we present the analysis of the BrowserID system. We first formulate fundamental security properties for the BrowserID system. We then present attacks that show that these properties are not satisfied and propose fixes. For the case of BrowserID with sIdP and the fixes applied, we then prove that the security properties are satisfied in our web model. We note that we also incorporate the automated CAP creation with the CIF in our model of BrowserID (see Section 4.2). Our web model is expressive enough to also formally model the BrowserID system with primary IdPs (and Sideshow/BigTent) in a straightforward way. However, we leave the detailed formulation of such a model and the proof of the security of the fixed system with *primary* IdPs to future work.

### 5.1 Security Properties for BrowserID

While the documentation of BrowserID does not contain explicit security goals, we deduce two fundamental security properties that can be informally described as follows (see Appendix H for a formal description): **(A)** *The attacker should not be able to use a service of RP as an honest user.* In other words, the attacker should not get hold of (be able to derive from his current knowledge) an RP service token for an ID of an honest user (browser), even if the browser was closed and then later used by a malicious user (i.e., after a `CLOSECORRUPT`). **(B)** *The attacker should not be able to authenticate an honest browser to an RP with an ID that is not owned by the browser.*

## 5.2 Attacks on BrowserID

Our analysis of BrowserID w.r.t. the above security properties revealed several attacks (as sketched next). We confirmed the attacks on the actual implementation and also reported them to Mozilla. The first two fixes proposed below have been adopted by Mozilla already and the others are currently under discussion at Mozilla.

### 5.2.1 Identity Forgery

There are two problems in Sideshow that lead to identity forgery attacks for Gmail addresses; analogously in BigTent with Yahoo email addresses.<sup>5</sup>

a) It is not checked if all requested attributes in the OpenID assertion are MACed, which allows for the following attack: A (web) attacker may choose any Gmail address to impersonate, say `victim@gmail.com`. He starts a BrowserID login with this email address. When he is then redirected to the OpenID URL at Gmail, he removes the email attribute from Sideshow's authentication request. The attacker authenticates himself at Gmail with his own account (say, `attacker@gmail.com`). Upon receipt of the OpenID assertion, he appends the email attribute with value `victim@gmail.com` and forwards it to Sideshow. The assertion is declared valid by Gmail since the MAC is correct (the email attribute is not listed in `openid.signed`). Since Sideshow does not require the email attribute to be in `openid.signed`, it accepts the OpenID assertion, considers the attacker's session to be authenticated for `victim@gmail.com`, and issues UCs for this address to the attacker. This violates Condition (A).

b) Sideshow uses the first email address in the OpenID assertion (based on the attribute type information), which is not necessarily the MACed email address. This allows for an attack similar to the above, except that the attacker does not need to change Sideshow's authentication request but only prepends the victim's email address to the OpenID assertion in an additional attribute.

**Proposed fix.** Sideshow/BigTent must ensure to use the correct and MACed attribute for the email address.

### 5.2.2 Login Injection Attack

During the login process, the origin of the response `postMessage` ([26] in Figure 4), which contains the CAP, is not checked. An attacker (e.g., in a malicious advertisement iframe within RP-Doc or in a parent window of RP-Doc), can continuously send `postMessages` to the RP-Doc with his own CAP in order to log the user into his own account. The outcome of this attack is similar to session swapping. For example, if the attacker is able to log the user into a search engine, the attacker might be able to read the search terms the user enters. This attack violates Condition (B).<sup>6</sup>

**Proposed fix.** To fix the problem, the sender's origin of the `postMessage` [26] must be checked to match LPO.

---

<sup>5</sup>See [https://bugzilla.mozilla.org/show\\_bug.cgi?id=920030](https://bugzilla.mozilla.org/show_bug.cgi?id=920030) and [https://bugzilla.mozilla.org/show\\_bug.cgi?id=920301](https://bugzilla.mozilla.org/show_bug.cgi?id=920301).

<sup>6</sup>See [https://bugzilla.mozilla.org/show\\_bug.cgi?id=868967](https://bugzilla.mozilla.org/show_bug.cgi?id=868967)

### 5.2.3 Key Cleanup Failure Attack

When LD creates a key pair ([15] in Figure 4), it stores the keys in the localStorage (even in ephemeral sessions). When a user quits a session (e.g. by clicking on RP’s logout button and closing the browser) the key pair (and the UC) remain in the localStorage, unlike session cookies. Hence, users of shared terminals can read the localStorage (in our model, a CLOSECORRUPT allows an attacker to do this) and then, using the key pair and the UC, create valid CAPs to log in at any RP under the identity of the previous user, which violates Condition (A).<sup>7</sup>

**Proposed fix.** We propose to use the localStorage for this data only in non-ephemeral sessions.

### 5.2.4 Cookie Cleanup Failure Attack (for the case of secondary IdP only)

The LPO session cookie is not deleted when the browser is closed, even in ephemeral sessions and even if a user logged out at RP beforehand. (In our model, if the attacker issues a CLOSECORRUPT, he can therefore still access the LPO session cookie.) Hence, another user of the same browser could request new UCs for *any* ID registered at LPO for that user, and hence, log in at any RP under this ID, which violates Condition (A).<sup>8</sup>

**Proposed fix.** In ephemeral sessions, LPO should limit the cookie lifetime to the browser session.

## 5.3 Analysis of BrowserID with sIdP

We now present an overview of our formal model and analysis of BrowserID with sIdP. More details are presented in Appendix G. We consider ephemeral sessions (the default), which are supposed to last until the browser is closed. We assume that users are already registered at LPO, i.e., they have accounts at LPO with one or more email addresses registered in each account.

More specifically, we first model the BrowserID system as a web system (in the sense of Section 3), then precisely formalize the security properties already sketched in Section 5.1 in this model, and finally prove, for the BrowserID model with the fixes proposed in Section 5.2 applied (otherwise the proof would not go through), that these security properties are satisfied.

We call a web system  $\mathcal{BID} = (\mathcal{W}, \mathcal{S}, \text{script}, E_0)$  a *BrowserID web system* if it is of the form described in what follows.

The system  $\mathcal{W} = \text{Hon} \cup \text{Web} \cup \text{Net}$  consists of the (network) attacker process **attacker**, the web server for LPO, a finite set  $\mathbf{B}$  of web browsers, and a finite set  $\text{RP}$  of web servers for the relying parties, with  $\text{Hon} := \mathbf{B} \cup \text{RP} \cup \{\text{LPO}\}$ ,  $\text{Web} := \emptyset$ , and  $\text{Net} := \{\text{attacker}\}$ . DNS servers are assumed to be dishonest, and hence, are subsumed by **attacker**. More details on the processes in  $\mathcal{W}$  are provided below.

The set  $\mathcal{N}$  of nonces is partitioned into three sets, an infinite set  $N^{\mathcal{W}}$ , an infinite set  $K_{\text{private}}$ , and a finite set **Secrets**. The set  $N^{\mathcal{W}}$  is further partitioned into infinite sets of nonces, one set  $N^p \subseteq N^{\mathcal{W}}$  for every  $p \in \mathcal{W}$ .

The set **Addresses** contains for LPO, **attacker**, every relying party in  $\text{RP}$ , and every browser in  $\mathbf{B}$  one address each. By **addr** we denote the corresponding assignment from a process to its address. The set **Domains** contains one domain for LPO, one for every relying party in  $\text{RP}$ , and a finite set of domains for **attacker**. Each domain is assigned a fresh private key (a nonce). Additionally, LPO has a fresh signing key  $k^{\text{LPO}}$ , which it uses to create UCs.

<sup>7</sup>See <https://github.com/mozilla/browserid/issues/3770>

<sup>8</sup>See <https://github.com/mozilla/browserid/issues/3769>

Each browser  $b \in \mathbf{B}$  owns a finite set of secrets ( $\subseteq \text{Secrets}$ ) for LPO and each secret is assigned a finite set of email addresses (IDs) of the form  $\langle name, d \rangle$ , with  $name \in \mathbb{S}$  and  $d \in \text{Domains}$ , such that browsers have disjoint sets of secrets and secrets have disjoint sets of IDs. An ID  $i$  is owned by a browser  $b$  if the secret associated with  $i$  belongs to  $b$ .

The set  $\mathcal{S}$  contains four scripts, with their string representations defined by `script`: the honest scripts running in RP-Doc, CIF, and LD, respectively, and the malicious script  $R^{\text{att}}$  (see below for more details).

The set  $E_0$  contains only the trigger events specified in Definition 6.

Before we specify the processes in  $\mathcal{W}$ , we first note that a UC  $uc$  for a user  $u$  with email address  $i$  and public key (verification key)  $\text{pub}(k_u)$ , where  $k_u$  is the private key (signing key) of  $u$ , is modeled to be a message of the form  $uc = \text{sig}(\langle i, \text{pub}(k_u) \rangle, k^{\text{LPO}})$ , with  $k^{\text{LPO}}$  as defined above. An IA  $ia$  for an origin  $o$  (e.g.,  $\langle \text{example.com}, \mathbb{S} \rangle$ ) is a message of the form  $ia = \text{sig}(o, k_u)$ . Now, a CAP is of the form  $\langle uc, ia \rangle$ . Note that the time stamps are omitted both from the UC and the IA. This models that both certificates are valid indefinitely. In reality, as explained in Section 4, they are valid for a certain period of time, as indicated by the time stamps. So our modeling is a safe overapproximation.

We are now ready to define the processes in  $\mathcal{W}$  as well as the scripts in  $\mathcal{S}$  in more detail. We note that in Appendix G, we provide a detailed formal specification of the processes and scripts in the style of Algorithm 9.

All processes in  $\mathcal{W}$  contain in their initial states all public keys and the private keys of their respective domains (if any). We define  $I^p = \{\text{addr}(p)\}$  for all  $p \in \text{Hon}$ .

**Attacker.** The attacker process is a network attacker (see Section 3.1), who uses all addresses for sending and listening. All parties use the attacker as a DNS server. See Appendix G.1 for details.

**Browsers.** Each  $b \in \mathbf{B}$  is a web browser as defined in Section 3.4. The initial state contains all secrets owned by  $b$ , stored under the origin  $\langle \text{dom}(\text{LPO}), \mathbb{S} \rangle$  of LPO;  $sts$  is  $\langle \text{dom}(\text{LPO}) \rangle$ . See Appendix G.2 for details.

**LPO.** The initial state of LPO contains its signing key  $k^{\text{LPO}}$ , all secrets in  $\text{Secrets}$  and the corresponding IDs. The definition of  $R^{\text{LPO}}$  closely follows the description of LPO in Section 4.4. Sessions of LPO expire non-deterministically. UCs are signed using  $k^{\text{LPO}}$ . See Appendix G.3 for details.

**Relying Parties.** A relying party  $r \in \text{RP}$  is a web server. The definition of  $R^r$  follows the description in Section 4 and the security considerations in [24].<sup>9</sup> RP answers any GET request with the script `script_RP_index` (see below). When receiving an HTTPS POST message, RP checks (among others) if the message contains a valid CAP. If successful, RP responds with an *RP service token for ID  $i$*  of the form  $\langle n, i \rangle$ , where  $i \in \text{ID}$  is the ID for which the CAP was issued and  $n$  is a freshly chosen nonce. The RP  $r$  keeps a list of such tokens in its state. Intuitively, a client having such a token can use the service of  $r$  for ID  $i$ . See Appendix G.4 for details.

**BrowserID Scripts.** The set  $\mathcal{S}$  consists of the scripts  $R^{\text{att}}$ , `script_RP_index`, `script_LPO_cif` and `script_LPO_ld` with their string representations `att_script`, `script_RP_index`, `script_LPO_cif`, and `script_LPO_ld`. The latter two scripts (issued by LPO) are defined in a straightforward way following the implementation outlined in Section 4. The script `script_RP_index`

<sup>9</sup>Mozilla recommends to (1) protect against Cross-site Request Forgery ( $R^r$  checks the Origin header, which is always set in our model), (2) verify CAPs on the server (rather than in the browser), (3) check if the CAP is issued for the correct RP, and (4) verify SSL certificates.

(issued by RP) also includes the script that is (in reality) loaded from LPO. In particular, this script creates the CIF and the LD iframes/subwindows, whose contents (scripts) are loaded from LPO. See Appendix G.5 for details.

#### 5.4 Security of the Fixed System

We call a BrowserID web system  $\mathcal{BID}$  with the fixes proposed in Section 5.2 a *fixed BrowserID web system*. We now obtain the following theorem, which says that such a system satisfies the security properties (A) and (B).

**Theorem 1.** *Let  $\mathcal{BID}$  be a fixed BrowserID web system. Then,  $\mathcal{BID}$  is secure.*

The proof of this theorem is presented in Appendix I.

## 6 Related Work

Early work in the direction of formal web security analysis includes work by Kerschbaum [22], in which a Cross-Site Request Forgery protection proposal is formally analyzed using a simple model expressed using Alloy, a finite-state model checker [20].

In seminal work, Akhawe et al. [2] initiated a more general formal treatment of web security. Again the model was provided in the Alloy modeling language. Inspired by this work, Bansal et al. [5, 6] built the WebSpi model for the web infrastructure, which is encoded in the modeling language (a variant of the applied pi-calculus [1]) of ProVerif, a specialized tool for cryptographic protocol analysis [8]. Both models have successfully been applied to find attacks in standards and web applications.

We see our work as a complement to these models: On the one hand, the above models support (fully) automated analysis. On the other hand, our model is much more comprehensive and accurate, but not directly suitable for automation.<sup>10</sup> We think that, similarly to the area of cryptography, both approaches, automated analysis and manual analysis, are very valuable. Clearly, it is highly desirable to push automated analysis as much as possible, given that manual proofs are laborious and error-prone. Conversely, automated approaches may miss important problems due to the less accurate models they consider. Moreover, a “service” more comprehensive and accurate models provide, even if they are manually driven, is that they summarize and condense relevant aspects in the various standards and specifications for the web. As such, they are an important basis for the formal foundation and discourse on web security and can serve as reference models (for tool-supported analysis, web security researchers, for developers of web technologies and standards, and maybe for teaching basic web security concepts).

The BrowserID system has been analyzed before using the AuthScan tool developed by Bai et al. [4]. Their work focusses on the automated extraction of a model from a protocol implementation. Their analysis of BrowserID is not very detailed; only two rather trivial attacks

---

<sup>10</sup>The tool-based models are necessarily tailored to and limited by constraints of the tools. For example, models for Alloy are necessarily finite state. Terms (messages) need to be encoded in some way as they are not directly supported. Due to the analysis method employed in ProVerif, the WebSpi model is of a monotonic nature. For instance, cookies and localStorage entries can only be added, but not deleted or modified. Also, the number of cookies per request is limited. Several features (that have been crucial for the analysis of BrowserID) are not supported by the tool-based models, including the precise handling of windows, documents, and iframes as well as cross-document messaging (postMessages), and the ability for an attacker to take over a browser after it has been closed. Dealing with such features in an automated tool is indeed challenging.

are identified, for example, CAPs that are sent unencrypted can be replayed by the attacker to an RP. There is also work on the analysis of other web-based single sign-on systems, such as SAML-based single sign-on, OpenID, and OAuth (see, e.g., [3, 7, 11, 15, 17, 23, 27–30]). However, none of these works are based on a model of the web infrastructure.

In [16, 26, 27, 30], potentially problematic usage of postMessages and the OpenID interface are discussed. While very useful, these papers do not consider BrowserID or formal models, and they do not formalize security properties for web applications or establish formal security guarantees.

Bohannon and Pierce propose a formal model of a web browser core [9]. The scope and goal of the model is different to ours, but some mechanisms can be found in both models. Börger et al. present an approach for the analysis of web application frameworks, focussing on the server [10].

## 7 Conclusion

We presented an expressive model of the web infrastructure and web applications, the most comprehensive model for the web infrastructure to date. It contains many security-relevant features and is designed to closely mimic standards and specifications for the web. As such, it constitutes a solid basis for the analysis of a broad range of web standards and applications.

In our case study, we analyzed the BrowserID system, found several very critical attacks, proposed fixes, and proved the fixed system for the case of secondary IdP case secure w.r.t. the security properties we specified. The analysis of this system is out of the scope of other models for the web infrastructure.

As for future work, it is straightforward to incorporate further features, such as subdomains, cross-origin resource sharing, and finer-grained settings for cookie paths and domains, which we have left out mainly for brevity of presentation for now. Our model could serve as a basis and a reference for automated approaches, where one could try to extend the existing automated approaches or develop new ones (e.g., based on theorem provers, where higher accuracy is typically paid by more interaction). Finally, BrowserID is being used by more and more web sites and it will continue to be an interesting object of study. An obvious next step is to analyze BrowserID for the case of primary IdPs. The model is already expressive enough to carry out such an analysis. We also plan to apply our model to other web applications and web standards.

## Acknowledgement

The first author is supported by the *Studienstiftung des Deutschen Volkes* (German National Academic Foundation).

## A Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of our web model and our case study presented in the rest of the appendix.

### A.1 Notations

For a set  $s$  and a sequence  $t = \langle t_1, \dots, t_n \rangle$  we use  $t \subset^{\diamond} s$  to say that  $t_1, \dots, t_n \in s$ . We define  $x \in^{\diamond} t \iff \exists i : t_i = x$ . We write  $t +^{\diamond} y$  to denote the sequence  $\langle t_1, \dots, t_n, y \rangle$ . For a sequence  $t = \langle t_1, \dots, t_n \rangle$  we define  $|t| = n$ . If  $t$  is not a sequence, we set  $|t| = \diamond$ . For a finite set  $M$  with  $M = \{m_1, \dots, m_n\}$  we use  $\langle M \rangle$  to denote the term of the form  $\langle m_1, \dots, m_n \rangle$ . (The order of the elements does not matter; one arbitrary is chosen.)

We also use specific terms which we call dictionaries.

**Definition 7.** A dictionary over  $X$  and  $Y$  is a term of the form  $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$ , where  $k_1, \dots, k_n \in X$ ,  $v_1, \dots, v_n \in Y$ , and the keys  $k_1, \dots, k_n$  are unique, i.e.,  $\forall i \neq j : k_i \neq k_j$ . We call every term  $\langle k_i, v_i \rangle$ ,  $i \in \{1, \dots, n\}$ , an element of the dictionary with key  $k_i$  and value  $v_i$ . We often write  $[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]$  instead of  $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$ . We denote the set of all dictionaries over  $X$  and  $Y$  by  $[X \times Y]$ .

We note that the empty dictionary is equivalent to the empty sequence:  $[] = \langle \rangle$ . Figure 5 shows the short notation for dictionary operations that will be used when describing the browser atomic process. For a dictionary  $z = [k_1 : v_1, k_2 : v_2, \dots, k_n : v_n]$  we write  $k \in z$  to say that there exists  $i$  such that  $k = k_i$ . We write  $z[k_j] := v_j$  to extract elements. If  $k \notin z$ , we set  $z[k] := \langle \rangle$ .

$$[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n][k_i] = v_i \quad (8)$$

$$\begin{aligned} [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1}, \dots, k_n : v_n] - k_i = \\ [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1}, \dots, k_n : v_n] \end{aligned} \quad (9)$$

**Figure 5.** Dictionary operators with  $1 \leq i \leq n$ .

Given a term  $t = \langle t_1, \dots, t_n \rangle$ , we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection  $\pi_i$  for the integers  $i$  in the sequence. We call such a sequence a *pointer*:

**Definition 8.** A pointer is a sequence of non-negative integers.

We write  $\tau.\bar{p}$  for the application of the pointer  $\bar{p}$  to the term  $\tau$ . This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

**Example 2.** For the term  $\tau = \langle a, b, \langle c, d, \langle e, f \rangle \rangle \rangle$  and the pointer  $\bar{p} = \langle 3, 1 \rangle$ , the subterm of  $\tau$  at the position  $\bar{p}$  is  $c = \pi_1(\pi_3(\tau))$ . Also,  $\tau.3.\langle 3, 1 \rangle = \tau.3.\bar{p} = \tau.3.3.1 = e$ .

To improve readability, we try to avoid writing, e.g.,  $o.2$  or  $\pi_2(o)$  in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as  $\langle host, protocol \rangle$  and  $o$  is an *Origin* term, then we can write  $o.protocol$  instead of  $\pi_2(o)$  or  $o.2$ . See also Example 3.

In our pseudocode, we will write, for example,

**let**  $x, y$  **such that**  $\langle \text{Constant}, x, y \rangle \equiv t$  **if possible; otherwise** doSomethingElse

for some variables  $x, y$ , a string **Constant**, and some term  $t$  to express that  $x := \pi_2(t)$ , and  $y := \pi_3(t)$  if **Constant**  $\equiv \pi_1(t)$  and if  $|\langle \text{Constant}, x, y \rangle| = |t|$ , and that otherwise  $x$  and  $y$  are not set and doSomethingElse is executed.

## A.2 URLs

**Definition 9.** A URL is a term  $\langle \text{URL}, \text{protocol}, \text{host}, \text{path}, \text{params} \rangle$  with  $\text{protocol} \in \{\mathbf{P}, \mathbf{S}\}$  (for plain (HTTP) and secure (HTTPS)),  $\text{host} \in \text{Domains}$ ,  $\text{path} \in \mathbb{S}$  and  $\text{params} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ . The set of all valid URLs is  $\text{URLs}$ .

**Example 3.** For the URL  $u = \langle \text{URL}, a, b, c, d \rangle$ ,  $u.\text{protocol} = a$ . If, in the algorithm described later, we say  $u.\text{path} := e$  then  $u = \langle \text{URL}, a, b, c, e \rangle$  afterwards.

## A.3 Origins

**Definition 10.** An origin is a term of the form  $\langle \text{host}, \text{protocol} \rangle$  with  $\text{host} \in \text{Domains}$ ,  $\text{protocol} \in \{\mathbf{P}, \mathbf{S}\}$ . We write  $\text{Origins}$  for the set of all origins. See Example 1 for an example for an origin.

## A.4 Cookies

**Definition 11.** A cookie is a term of the form  $\langle \text{name}, \text{content} \rangle$  where  $\text{name} \in \mathcal{T}_{\mathcal{N}}$ , and  $\text{content}$  is a term of the form  $\langle \text{value}, \text{secure}, \text{session}, \text{httpOnly} \rangle$  where  $\text{value} \in \mathcal{T}_{\mathcal{N}}$ ,  $\text{secure}$ ,  $\text{session}$ ,  $\text{httpOnly} \in \{\top, \perp\}$ . We write  $\text{Cookies}$  for the set of all cookies.

Note that cookies of the form described here are only contained in HTTP(S) requests. In responses, only the components  $\text{name}$  and  $\text{value}$  are transferred as a pairing of the form  $\langle \text{name}, \text{value} \rangle$ .

## A.5 HTTP Messages

**Definition 12.** An HTTP request is a term of the form shown in (10). An HTTP response is a term of the form shown in (11).

$$\langle \text{HTTPReq}, \text{nonce}, \text{method}, \text{host}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \quad (10)$$

$$\langle \text{HTTPResp}, \text{nonce}, \text{status}, \text{headers}, \text{body} \rangle \quad (11)$$

The components are defined as follows:

- $\text{nonce} \in \mathcal{N}$  serves to map each response to the corresponding request
- $\text{method} \in \text{Methods}$  is one of the HTTP methods.
- $\text{host} \in \text{Domains}$  is the host name in the *HOST* header of HTTP/1.1.
- $\text{path} \in \mathbb{S}$  is a string indicating the requested resource at the server side
- $\text{status} \in \mathbb{S}$  is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard)
- $\text{parameters} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$  contains URL parameters
- $\text{headers} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ , containing request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \text{Origin}, o \rangle$  where  $o$  is an origin
  - $\langle \text{Set-Cookie}, c \rangle$  where  $c$  is a sequence of cookies

- $\langle \text{Cookie}, c \rangle$  where  $c \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$  (note that in this header, only names and values of cookies are transferred)
- $\langle \text{Location}, l \rangle$  where  $l \in \text{URLs}$
- $\langle \text{Strict-Transport-Security}, \top \rangle$
- $\text{body} \in \mathcal{T}_{\mathcal{N}}$  in requests and responses.

We write  $\text{HTTPRequests}/\text{HTTPResponses}$  for the set of all HTTP requests or responses, respectively.

**Example 4** (HTTP Request and Response).

$$r := \langle \text{HTTPReq}, n_1, \text{POST}, \text{example.com}, /show, \langle \langle \text{index}, 1 \rangle \rangle, [\text{Origin} : \langle \text{example.com}, \mathbb{S} \rangle], \langle \text{foo}, \text{bar} \rangle \rangle \quad (12)$$

$$s := \langle \text{HTTPResp}, n_1, 200, \langle \langle \text{Set-Cookie}, \langle \langle \text{SID}, \langle n_2, \perp, \perp, \top \rangle \rangle \rangle \rangle \rangle, \langle \text{somescript}, x \rangle \rangle \quad (13)$$

An HTTP GET request for the URL <http://example.com/show?index=1> is shown in (12), with an Origin header and a body that contains  $\langle \text{foo}, \text{bar} \rangle$ . A possible response is shown in (13), which contains an `httpOnly` cookie with name `SID` and value  $n_2$  as well as the string representation `somescript` of the scripting process  $\text{script}^{-1}(\text{somescript})$  (which should be an element of  $\mathcal{S}$ ) and its initial state  $x$ .

### Encrypted HTTP Messages

For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supported to encrypt the response using the symmetric key.

**Definition 13.** An encrypted HTTP request is of the form  $\text{enc}_a(\langle m, k' \rangle, k)$ , where  $k, k' \in \mathcal{K}$  and  $m \in \text{HTTPRequests}$ . The corresponding encrypted HTTP response would be of the form  $\text{enc}_s(m', k')$ , where  $m' \in \text{HTTPResponses}$ . We call the sets of all encrypted HTTP requests and responses  $\text{HTTPSRequests}$  or  $\text{HTTPSResponses}$ , respectively.

**Example 5.**

$$\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}})) \quad (14)$$

$$\text{enc}_s(s, k') \quad (15)$$

The term (14) shows an encrypted request (with  $r$  as in (12)). It is encrypted using the public key  $\text{pub}(k_{\text{example.com}})$ . The term (15) is a response (with  $s$  as in (13)). It is encrypted symmetrically using the (symmetric) key  $k'$  that was sent in the request (14).

### A.6 DNS Messages

**Definition 14.** A DNS request is a term of the form  $\langle \text{DNSResolve}, \text{domain}, \text{nonce} \rangle$  where  $\text{domain} \in \text{Domains}$ ,  $\text{nonce} \in \mathcal{N}$ . We call the set of all DNS requests  $\text{DNSRequests}$ .

**Definition 15.** A DNS response is a term of the form  $\langle \text{DNSResolved}, \text{result}, \text{nonce} \rangle$  with  $\text{result} \in \text{Addresses}$ ,  $\text{nonce} \in \mathcal{N}$ . We call the set of all DNS responses  $\text{DNSResponses}$ .

As already mentioned in Section 3.2, DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back.

## B Detailed Description of the Browser Model

Following the informal description of the browser model in Section 3.4, we now present a formal model. We start by introducing some notation and terminology.

### B.1 Notation and Terminology (Web Browser State)

**Definition 16.** A window is a term of the form  $w = \langle \text{nonce}, \text{documents}, \text{opener} \rangle$  with  $\text{nonce} \in \mathcal{N}$ ,  $\text{documents} \subset^{\diamond} \text{Documents}$  (defined below),  $\text{opener} \in \mathcal{N} \cup \{\perp\}$  where  $d.\text{active} = \top$  for exactly one  $d \in^{\diamond} \text{documents}$  if  $\text{documents}$  is not empty (we then call  $d$  the active document of  $w$ ). We write  $\text{Windows}$  for the set of all windows. We write  $w.\text{activedocument}$  to denote the active document inside window  $w$  if it exists and  $\langle \rangle$  else.

We will refer to the window nonce as (*window*) reference. See Example 1 for an example of a window.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the “back” button) and the documents in the window term to the right of the currently active document are documents available via the “forward” button, as will be clear from the description of web browser model (see Section B.2).

A window  $a$  may have opened a top-level window  $b$  (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term  $b$  is the nonce of  $a$ , i.e.,  $b.\text{opener} = a.\text{nonce}$ .

**Definition 17.** A document  $d$  is a term of the form

$$\langle \text{nonce}, \text{origin}, \text{script}, \text{scriptstate}, \text{scriptinput}, \text{subwindows}, \text{active} \rangle$$

where  $\text{nonce} \in \mathcal{N}$ ,  $\text{origin} \in \text{Origins}$ ,  $\text{script}, \text{scriptstate}, \text{scriptinput} \in \mathcal{T}_{\mathcal{N}}$ ,  $\text{subwindows} \subset^{\diamond} \text{Windows}$ ,  $\text{active} \in \{\top, \perp\}$ . A limited document is a term of the form  $\langle \text{nonce}, \text{subwindows} \rangle$  with  $\text{nonce}, \text{subwindows}$  as above. A window  $w \in^{\diamond} \text{subwindows}$  is called a subwindow (of  $d$ ). We write  $\text{Documents}$  for the set of all documents.

We will refer to the document nonce as (*document*) reference. An example for window and document terms was given in Example 1.

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 7.

**Definition 18.** Let  $OR := \{\langle o, r \rangle \mid o \in \text{Origins}, r \in \mathcal{N}\}$ . The set of states  $Z^p$  of a web browser atomic process  $p$  consists of the terms of the form

$$\langle \text{windows}, \text{secrets}, \text{cookies}, \text{localStorage}, \text{sessionStorage}, \text{keyMapping}, \text{sts}, \text{DNSaddress}, \text{nonces}, \text{pendingDNS}, \text{pendingRequests}, \text{isCorrupted} \rangle$$

where  $\text{windows} \subset^{\diamond} \text{Windows}$ ,  $\text{secrets} \in [\text{Origins} \times \mathcal{N}]$ ,  $\text{cookies}$  is a dictionary over  $\text{Domains}$  and dictionaries of  $\text{Cookies}$ ,  $\text{localStorage} \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$ ,  $\text{sessionStorage} \in [OR \times \mathcal{T}_{\mathcal{N}}]$ ,  $\text{keyMapping} \in [\text{Domains} \times \mathcal{N}]$ ,  $\text{sts} \subset^{\diamond} \text{Domains}$ ,  $\text{DNSaddress} \in \text{Addresses}$ ,  $\text{nonces} \subset^{\diamond} \mathcal{N}$ ,  $\text{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ ,  $\text{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$ ,  $\text{isCorrupted} \in \{\perp, \text{FULLCORRUPT}, \text{CLOSECORRUPT}\}$ . We call the set of all states of standard HTTP browsers  $\text{SHBStates}$ .

**Definition 19.** For two window terms  $w$  and  $w'$  we write  $w \xrightarrow{\text{childof}} w'$  if  $w \in \langle w'.\text{activedocument.subwindows} \rangle$ . We write  $\xrightarrow{\text{childof}^+}$  for the transitive closure.

In the following description of the web browser relation  $R^P$  we will use the helper functions `Subwindows`, `Docs`, `Clean`, `CookieMerge` and `AddCookie`.

Given a browser state  $s$ , `Subwindows(s)` denotes the set of all pointers<sup>11</sup> to windows in the window list  $s.\text{windows}$ , their (active) documents, and the subwindows of these documents (recursively). We exclude subwindows of inactive documents and their subwindows. With `Docs(s)` we denote the set of pointers to all active documents in the set of windows referenced by `Subwindows(s)`.

**Definition 20.** For a browser state  $s$  we denote by `Subwindows(s)` the minimal set of pointers that satisfies the following conditions: (1) For all windows  $w \in \langle s.\text{windows} \rangle$  there is a  $\bar{p} \in \text{Subwindows}(s)$  such that  $s.\bar{p} = w$ . (2) For all  $\bar{p} \in \text{Subwindows}(s)$ , the active document  $d$  of the window  $s.\bar{p}$  and every subwindow  $w$  of  $d$  there is a pointer  $\bar{p}' \in \text{Subwindows}(s)$  such that  $s.\bar{p}' = w$ .

Given a browser state  $s$ , the set `Docs(s)` of pointers to active documents is the minimal set such that for every  $\bar{p} \in \text{Subwindows}(s)$ , there is a pointer  $\bar{p}' \in \text{Docs}(s)$  with  $s.\bar{p}' = s.\bar{p}.\text{activedocument}$  ■

The function `Clean` will be used to determine which information about windows and documents the script running in the document  $d$  has access to.

**Definition 21.** Let  $s$  be a browser state and  $d$  a document. By `Clean(s, d)` we denote the term that equals  $s.\text{windows}$  but with all inactive documents removed (including their subwindows etc.) and all subterms that represent non-same-origin documents w.r.t.  $d$  replaced by a limited document  $d'$  with the same nonce and the same subwindow list. Note that non-same-origin documents on all levels are replaced by their corresponding limited document.

The function `CookieMerge` merges two sequences of cookies together: When used in the browser, `oldcookies` is the sequence of existing cookies for some origin, `newcookies` is a sequence of new cookies that was outputted by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in RFC6265.

**Definition 22.** The set `CookieMerge(oldcookies, newcookies)` for two sequences `oldcookies` and `newcookies` of cookies (where the cookies in `oldcookies` have pairwise different names) is defined by the following algorithm: From `newcookies` remove all cookies  $c$  that have  $c.\text{content.httpOnly} \equiv \top$ . For any  $c, c' \in \langle \text{newcookies} \rangle$ ,  $c.\text{name} \equiv c'.\text{name}$ , remove the cookie that appears left of the other in `newcookies`. Let  $m$  be the set of cookies that have a name that either appears in `oldcookies` or in `newcookies`, but not in both. For all pairs of cookies  $(c_{\text{old}}, c_{\text{new}})$  with  $c_{\text{old}} \in \langle \text{oldcookies} \rangle$ ,  $c_{\text{new}} \in \langle \text{newcookies} \rangle$ ,  $c_{\text{old}}.\text{name} \equiv c_{\text{new}}.\text{name}$ , add  $c_{\text{new}}$  to  $m$  if  $c_{\text{old}}.\text{content.httpOnly} \equiv \perp$  and add  $c_{\text{old}}$  to  $m$  otherwise. The result of `CookieMerge(oldcookies, newcookies)` is  $m$ .

The function `AddCookie` adds a cookie  $c$  to the sequence of cookies contained in the sequence `oldcookies`. It is again based on the algorithm described in RFC6265 but simplified for the use in the browser model.

**Definition 23.** The sequence `AddCookie(oldcookies, c)`, where `oldcookies` is a sequence of cookies with different names and  $c$  is a cookie  $c$ , is defined by the following algorithm: Let  $m := \text{oldcookies}$ . Remove any  $c'$  from  $m$  that has  $c.\text{name} \equiv c'.\text{name}$ . Append  $c$  to  $m$  and return  $m$ .

<sup>11</sup>Recall the definition of a pointer from Definition 8.

The function `NavigableWindows` returns a set of windows that a document is allowed to navigate. We closely follow [18], Section 5.1.4 for this definition.

**Definition 24.** `NavigableWindows( $\bar{w}, s'$ )` is the set  $\bar{W} \subseteq \text{Subwindows}(s')$  of pointers to windows that the active document in  $\bar{w}$  is allowed to navigate. The set  $\bar{W}$  is defined to be the minimal set such that for every  $\bar{w}' \in \text{Subwindows}(s')$  the following is true:

- If  $s'.\bar{w}'.\text{activedocument.origin} \equiv s'.\bar{w}.\text{activedocument.origin}$  (the active documents in  $\bar{w}$  and  $\bar{w}'$  are same-origin), then  $\bar{w}' \in \bar{W}$ , and
- If  $s'.\bar{w} \xrightarrow{\text{childof}^*} s'.\bar{w}' \wedge \nexists \bar{w}'' \in \text{Subwindows}(s')$  with  $s'.\bar{w}' \xrightarrow{\text{childof}^*} s'.\bar{w}''$  ( $\bar{w}'$  is a top-level window and  $\bar{w}$  is an ancestor window of  $\bar{w}'$ ), then  $\bar{w}' \in \bar{W}$ , and
- If  $\exists \bar{p} \in \text{Subwindows}(s')$  such that  $s'.\bar{w}' \xrightarrow{\text{childof}^+} s'.\bar{p} \wedge s'.\bar{p}.\text{activedocument.origin} = s'.\bar{w}.\text{activedocument.origin}$  ( $\bar{w}'$  is not a top-level window but there is an ancestor window  $\bar{p}$  of  $\bar{w}'$  with an active document that has the same origin as the active document in  $\bar{w}$ ), then  $\bar{w}' \in \bar{W}$ , and
- If  $\exists \bar{p} \in \text{Subwindows}(s')$  such that  $s'.\bar{w}'.\text{opener} = s'.\bar{p}.\text{nonce} \wedge \bar{p} \in \bar{W}$  ( $\bar{w}'$  is a top-level window—it has an opener—and  $\bar{w}$  is allowed to navigate the opener window of  $\bar{w}'$ ,  $\bar{p}$ ), then  $\bar{w}' \in \bar{W}$ .

## B.2 Description of the Web Browser Atomic Process

We will now describe the relation  $R^p$  of a standard HTTP browser  $p$ . For a tuple  $r = (((a:f:m), s), (M, s'))$  we define  $r$  to belong to  $R^p$  iff the non-deterministic algorithm presented in Section B.2.2, when given  $((a:f:m), s)$  as input, terminates with **stop**  $M, s'$ , i.e., with output  $M$  and  $s'$ . Recall that  $(a:f:m)$  is an (input) event and  $s$  is a (browser) state,  $M$  is a set of (output) events, and  $s'$  is a new (browser) state.

The notation **let**  $n \leftarrow N$  is used to describe that  $n$  is chosen non-deterministically from the set  $N$ . We write **for each**  $s \in M$  **do** to denote that the following commands (until **end for**) are repeated for every element in  $M$ , where the variable  $s$  is the current element. The order in which the elements are processed is chosen non-deterministically.

We first define some functions which will be used in the main algorithm presented in Section B.2.2.

### B.2.1 Functions

In the description of the following functions we use  $a, f, m, s$  and  $N^p$  as read-only global input variables. Also, the functions use the set  $N^p$  as a read-only set. All other variables are local variables or arguments.

`TAKENONCE` returns a nonce from the set of unused nonces and modifies the browser state such that the nonce is added to the sequence of used nonces. Note that this function returns two values, the nonce  $n$  and the modified state  $s'$ .

---

**Algorithm 1** Non-deterministically choose a fresh nonce.

---

- 1: **function** `TAKENONCE( $s'$ )`
- 2:   **let**  $n \leftarrow \{x \mid x \in N^p \wedge x \notin s'.\text{nonces}\}$

```

3:   let  $s'.nonces := s'.nonces + \langle \rangle n$ 
4:   return  $n, s'$ 
5: end function

```

The following function, GETNAVIGABLEWINDOW, is called by the browser to determine the window that is *actually* navigated when a script in the window  $s'.\bar{w}$  provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce)  $window$ , GETNAVIGABLEWINDOW returns a pointer to a selected window term in  $s'$ :

- If  $window$  is the string `_BLANK`, a new window is created and a pointer to that window is returned.
- If  $window$  is a nonce (reference) and there is a window term with a reference of that value in the windows in  $s'$ , a pointer  $\bar{w}'$  to that window term is returned, as long as the window is navigable by the current window's document (as defined by `NavigableWindows` above).

In all other cases,  $\bar{w}$  is returned instead (the script navigates its own window).

---

**Algorithm 2** Determine window for navigation.

---

```

1: function GETNAVIGABLEWINDOW( $\bar{w}, window, s'$ )
2:   if  $window \equiv \_BLANK$  then ▷ Open a new window when \_BLANK is used
3:     let  $n, s' := TAKENONCE(s')$ 
4:     let  $w' := \langle n, \langle \rangle, s'.\bar{w}.nonce$ 
5:     let  $s'.windows := s'.windows + \langle \rangle w'$  and let  $\bar{w}'$  be a pointer to this new element in  $s'$ 
6:     return  $(\bar{w}', s')$ 
7:   end if
8:   let  $\bar{w}' \leftarrow \text{NavigableWindows}(\bar{w}, s')$  such that  $s'.\bar{w}'.nonce \equiv window$  if possible; otherwise
   return  $(\bar{w}, s')$ 
9:   return  $(\bar{w}', s')$ 
10: end function

```

The following function takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.

---

**Algorithm 3** Determine same-origin window.

---

```

1: function GETWINDOW( $\bar{w}, window, s'$ )
2:   let  $\bar{w}' \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}'.nonce \equiv window$  if possible; otherwise return
    $(\bar{w}, s')$ 
3:   if  $s'.\bar{w}'.activedocument.origin \equiv s'.\bar{w}.activedocument.origin$  then
4:     return  $(\bar{w}', s')$ 
5:   end if
6:   return  $(\bar{w}, s')$ 
7: end function

```

The next function is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference  $n$ .

---

**Algorithm 4** Cancel pending requests for given window.

---

```

1: function CANCELNAV( $n, s'$ )
2:   remove all  $\langle n, req, key, f \rangle$  from  $s'.pendingRequests$  for any  $req, key, f$ 
3:   remove all  $\langle x, \langle n, message, protocol \rangle \rangle$  from  $s'.pendingDNS$  for any  $x, message, protocol$ 

```

4: **return**  $s'$   
5: **end function**

The following function takes an HTTP request  $message$  as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in  $s'.pendingDNS$  until the DNS resolution finishes. For normal HTTP requests,  $reference$  is a window reference. For XHRs,  $reference$  is a value of the form  $\langle document, nonce \rangle$  where  $document$  is a document reference and  $nonce$  is some nonce that was chosen by the script that initiated the request.  $protocol$  is either P or S.  $origin$  is the origin header value that is to be added to the HTTP request.

---

**Algorithm 5** Prepare headers, do DNS resolution, save message.

---

```

1: function SEND( $reference, message, protocol, origin, s'$ )
2:   if  $message.host \in \langle \rangle s'.sts$  then
3:     let  $protocol := S$ 
4:   end if
5:   let  $cookies := \{\langle c.name, c.content.value \rangle | c \in \langle \rangle s'.cookies[message.host]$ 
       $\wedge (c.content.secure \implies (protocol = S))\}$ 
6:   let  $message.headers[Cookie] := cookies$ 
7:   if  $origin \neq \perp$  then
8:     let  $message.headers[Origin] := origin$ 
9:   end if
10:  let  $n, s' := TAKENONCE(s')$ 
11:  let  $s'.pendingDNS[n] := \langle reference, message, protocol \rangle$ 
12:  stop  $\{(s'.DNSaddress : a : \langle DNSResolve, host, n \rangle)\}, s'$ 
13: end function

```

The following two functions have informally been described in Section 3.4.2.

The function RUNSCRIPT performs a script execution step of the script in the document  $s'.\bar{d}$  (which is part of the window  $s'.\bar{w}$ ). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted. Note that **for each** (Line 13) works in a non-deterministic order.

---

**Algorithm 6** Execute a script.

---

```

1: function RUNSCRIPT( $\bar{w}, \bar{d}, s'$ )
2:   let  $n, s' := TAKENONCE(s')$ 
3:   let  $tree := Clean(s', s'.\bar{d})$ 
4:   let  $cookies := \{\langle c.name, c.content.value \rangle | c \in \langle \rangle s'.cookies[s'.\bar{d}.origin.host]$ 
       $\wedge c.content.httpOnly = \perp$ 
       $\wedge (c.content.secure \implies (s'.\bar{d}.origin.protocol \equiv S))\}$ 
5:   let  $tlw \leftarrow s'.windows$  such that  $tlw$  is the top-level window containing  $d$ 
6:   let  $sessionStorage := s'.sessionStorage[\langle s'.\bar{d}.origin, tlw.nonce \rangle]$ 
7:   let  $localStorage := s'.localStorage[s'.\bar{d}.origin]$ 
8:   let  $secret := s'.secrets[s'.\bar{d}.origin]$ 
9:   let  $nonces$  be an infinite subset of  $\{x | x \in N^p \wedge x \notin \langle \rangle s'.nonces\}$ 
10:  let  $R \leftarrow script^{-1}(s'.\bar{d}.script)$ 
11:  let  $in :=$ 
       $\langle tree, s'.\bar{d}.nonce, s'.\bar{d}.scriptstate, s'.\bar{d}.scriptinput, cookies, localStorage, sessionStorage secret \rangle$ 
12:  let  $state' \leftarrow \mathcal{I}_{\mathcal{N}}, cookies' \leftarrow Cookies, localStorage' \leftarrow \mathcal{I}_{\mathcal{N}}, command \leftarrow \mathcal{I}_{\mathcal{N}}, out :=$ 
       $\langle state', cookies', localStorage', sessionStorage', command \rangle$  such that  $((in, nonces), out) \in R$ 

```

```

13:   for each  $n \in d_{\mathcal{N}}(\langle in, out \rangle) \cap NP$  do
14:     let  $s'.nonces := s'.nonces + \langle \rangle n$ 
15:   end for
16:   let  $s'.cookies[s'.\bar{d}.origin.host] := \langle CookieMerge(s'.cookies[s'.\bar{d}.origin.host], cookies') \rangle$ 
17:   let  $s'.localStorage[s'.\bar{d}.origin] := localStorage'$ 
18:   let  $s'.sessionStorage[\langle s'.\bar{d}.origin, thw.nonice \rangle] := sessionStorage'$ 
19:   let  $s'.\bar{d}.scriptstate := state'$ 
20:   switch command do
21:     case  $\langle \rangle$ 
22:       stop  $\{\}, s'$ 
23:     case  $\langle HREF, url, hrefwindow \rangle$ 12
24:       let  $\bar{w}', s' := GETNAVIGABLEWINDOW(\bar{w}, hrefwindow, s')$ 
25:       let  $req := \langle GET, url.host, HTTPReq, n, url.path, \langle \rangle, url.params, \langle \rangle \rangle$ 
26:       let  $s' := CANCELNAV(s'.\bar{w}'.nonce, s')$ 
27:       SEND $(s'.\bar{w}'.nonce, req, url.protocol, \perp, s')$ 
28:     case  $\langle IFRAME, url, window \rangle$ 
29:       let  $\bar{w}', s' := GETWINDOW(\bar{w}, window, s')$ 
30:       let  $req := \langle GET, url.host, HTTPReq, n, url.path, \langle \rangle, url.params, \langle \rangle \rangle$ 
31:       let  $n, s' := TAKENONCE(s')$ 
32:       let  $w' := \langle n, \langle \rangle, \perp \rangle$ 
33:       let  $s'.\bar{w}'.activedocument.subwindows := s'.\bar{w}'.activedocument.subwindows + \langle \rangle w'$ 
34:       SEND $(n, req, url.protocol, \perp, s')$ 
35:     case  $\langle FORM, url, method, data, hrefwindow \rangle$ 
36:       if  $method \notin \{GET, POST\}$  then13
37:         stop  $\{\}, s'$ 
38:       end if
39:       let  $\bar{w}', s' := GETNAVIGABLEWINDOW(\bar{w}, hrefwindow, s')$ 
40:       if  $method = GET$  then
41:         let  $body := \langle \rangle$ 
42:         let  $params := data$ 
43:         let  $origin := \perp$ 
44:       else
45:         let  $body := data$ 
46:         let  $params := url.params$ 
47:         let  $origin := s'.\bar{d}.origin$ 
48:       end if
49:       let  $req := \langle method, url.host, HTTPReq, n, url.path, \langle \rangle, params, body \rangle$ 
50:       let  $s' := CANCELNAV(s'.\bar{w}'.nonce, s')$ 
51:       SEND $(s'.\bar{w}'.nonce, req, url.protocol, origin, s')$ 
52:     case  $\langle SETSCRIPT, window, script \rangle$ 
53:       let  $\bar{w}', s' := GETWINDOW(\bar{w}, window, s')$ 
54:       let  $s'.\bar{w}'.activedocument.script := script$ 
55:       stop  $\{\}, s'$ 
56:     case  $\langle SETSCRIPTSTATE, window, scriptstate \rangle$ 
57:       let  $\bar{w}', s' := GETWINDOW(\bar{w}, window, s')$ 
58:       let  $s'.\bar{w}'.activedocument.scriptstate := scriptstate$ 
59:       stop  $\{\}, s'$ 

```

<sup>12</sup>See the definition of URLs in Appendix A.2.

<sup>13</sup>The working draft for HTML5 allowed for DELETE and PUT methods in HTML5 forms. However, these have since been removed. See <http://www.w3.org/TR/2010/WD-html5-diff-20101019/#changes-2010-06-24>.

```

60:   case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
61:     if method ∈ {CONNECT, TRACE, TRACK} then 14
62:       stop {}, s'
63:     end if
64:     if url.host ≠ s'.d̄.origin.host ∨ url.protocol ≠ s'.d̄.origin.protocol then 15
65:       stop {}, s'
66:     end if
67:     if method ∈ {GET, HEAD} then
68:       let data := ⟨⟩
69:       let origin := ⊥
70:     else
71:       let origin := s'.d̄.origin
72:     end if
73:     let req := ⟨method, url.host, HTTPReq, n, url.path, , url.params, data⟩
74:     SEND(⟨s'.d̄.nonce, xhrreference⟩, req, url.protocol, origin, s')
75:   case ⟨BACK, window⟩ 16
76:     let w̄', s' := GETNAVIGABLEWINDOW(w̄, window, s')
77:     if ∃ j̄ ∈ ℕ, j̄ > 1 such that s'.w̄'.documents.j̄.active ≡ ⊤ then
78:       let s'.w̄'.documents.j̄.active := ⊥
79:       let s'.w̄'.documents.(j̄ - 1).active := ⊤
80:       let s' := CANCELNAV(s'.w̄'.nonce, s')
81:     end if
82:     stop {}, s'
83:   case ⟨FORWARD, window⟩
84:     let w̄', s' := GETNAVIGABLEWINDOW(w̄, window, s')
85:     if ∃ j̄ ∈ ℕ such that s'.w̄'.documents.j̄.active ≡ ⊤ ∧
86:       s'.w̄'.documents.(j̄ + 1) ∈ Documents then
87:       let s'.w̄'.documents.j̄.active := ⊥
88:       let s'.w̄'.documents.(j̄ + 1).active := ⊤
89:       let s' := CANCELNAV(s'.w̄'.nonce, s')
90:     end if
91:     stop {}, s'
92:   case ⟨CLOSE, window⟩
93:     let w̄', s' := GETNAVIGABLEWINDOW(w̄, window, s')
94:     remove s'.w̄' from the sequence containing it
95:     stop {}, s'
96:   case ⟨POSTMESSAGE, window, message, origin⟩
97:     let w̄' ← Subwindows(s') such that s'.w̄'.nonce ≡ window
98:     if ∃ j̄ ∈ ℕ such that s'.w̄'.documents.j̄.active ≡ ⊤ ∧
99:       (origin ≠ ⊥ ⇒ s'.w̄'.documents.j̄.origin ≡ origin) then
100:       let s'.w̄'.documents.j̄.scriptinput := s'.w̄'.documents.j̄.scriptinput + ⟨⟩
101:       ⟨POSTMESSAGE, s'.w̄'.nonce, s'.d̄.origin, message⟩
102:     end if
103: end function

```

The function `PROCESSRESPONSE` is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. In *reference*, either

<sup>14</sup>According to W3C XMLHttpRequest definition.

<sup>15</sup>We only allow same origin requests for now.

<sup>16</sup>Note that navigating a window using the back/forward buttons does not trigger a reload of the affected documents. While real world browser may chose to refresh a document in this case, we assume that the complete state of a previously viewed document is restored.

a window or a document reference is given (see explanation for Algorithm 5 above). Again, *protocol* is either P or S.

The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response to the respective receiver (for XHR responses).

---

**Algorithm 7** Process an HTTP response.

---

```

1: function PROCESSRESPONSE(response, reference, request, protocol, s')
2:   let n, s' := TAKENONCE(s')
3:   if Set-Cookie ∈ response.headers then
4:     for each c ∈ ⟨ response.headers[Set-Cookie], c ∈ Cookies do
5:       let s'.cookies[request.url.host] := AddCookie(s'.cookies[request.url.host], c)
6:     end for
7:   end if
8:   if Strict-Transport-Security ∈ response.headers ∧ protocol ≡ S then
9:     let s'.sts := s'.sts + ⟨ request.host
10:  end if
11:  if Location ∈ response.headers ∧ response.status ∈ {303, 307} then 17
12:    let url := response.headers[Location]
13:    let method' := request.method 18
14:    let body' := request.body 19
15:    if Origin ∈ request.headers then
16:      let origin := ⟨request.headers[Origin], ⟨request.host, protocol⟩⟩
17:    else
18:      let origin := ⊥
19:    end if
20:    if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
21:      let method' := GET
22:      let body' := ⟨⟩
23:    end if
24:    if ∄  $\bar{w} \in \text{Subwindows}(s')$  such that s'. $\bar{w}$ .nonce ≡ reference then    ▷ Do not redirect XHRs.
25:      stop {}, s
26:    end if
27:    let req := ⟨method', url.host, HTTPReq, n, url.path, ⟨⟩, url.params, body'⟩
28:    SEND(reference, req, url.protocol, origin, s')
29:  end if
30:  if ∃  $\bar{w} \in \text{Subwindows}(s')$  such that s'. $\bar{w}$ .nonce ≡ reference then    ▷ normal response
31:    let script :=  $\pi_1(\text{response.body})$ 
32:    let scriptstate :=  $\pi_2(\text{response.body})$ 

```

---

<sup>17</sup>The RFC for HTTPbis (currently in draft status), which obsoletes RFC 2616, does not specify whether a POST/DELETE/etc. request that was answered with a status code of 301 or 302 should be rewritten to a GET request or not (“for historic reasons” that are detailed in Section 7.4). As the specification is clear for the status codes 303 and 307 (and most browsers actually follow the specification in this regard), we focus on modeling these.

<sup>18</sup>While the standard demands that users confirm redirections of non-safe-methods (e.g., POST), we assume that users generally confirm these redirections.

<sup>19</sup>If, for example, a GET request is redirected and the original request contained a body, this body is preserved, as HTTP allows for payloads in messages with all HTTP methods, except for the TRACE method (a detail which we omit). Browsers will usually not send body payloads for methods that do not specify semantics for such data in the first place.

```

33:   let  $d := \langle n, \langle request.host, request.protocol \rangle, script, scriptstate, \langle \rangle, \langle \rangle, \top \rangle$ 
34:   if  $s'.\bar{w}.documents \equiv \langle \rangle$  then
35:     let  $s'.\bar{w}.documents := \langle d \rangle$ 
36:   else
37:     let  $\bar{i} \leftarrow \mathbb{N}$  such that  $s'.\bar{w}.documents.\bar{i}.active \equiv \top$ 
38:     let  $s'.\bar{w}.documents.\bar{i}.active := \perp$ 
39:     remove  $s'.\bar{w}.documents.(\bar{i} + 1)$  and all following documents from  $s'.\bar{w}.documents$ 
40:     let  $s'.\bar{w}.documents := s'.\bar{w}.documents +^{\langle \rangle} d$ 
41:   end if
42:   stop  $\{\}, s'$ 
43: else if  $\exists \bar{w} \in \text{Subwindows}(s'), \bar{d}$  such that  $s'.\bar{d}.nonce \equiv \pi_1(\text{reference}) \wedge$ 
       $s'.\bar{d} = s'.\bar{w}.activedocument$  then ▷ process XHR response
44:   let  $s'.\bar{d}.scriptinput := s'.\bar{d}.scriptinput +^{\langle \rangle} \langle \text{XMLHTTPREQUEST}, response.body, \pi_2(\text{reference}) \rangle$ 
45: end if
46: end function

```

### B.2.2 Main Algorithm

This is the main algorithm of the browser relation. It was already presented informally in Section 3.4 and follows the structure presented there. It receives the message  $m$  as input, as well as  $a$ ,  $f$  and  $s$  as above.

---

#### Algorithm 8 Main Algorithm

---

```

Input:  $(a:f:m), s$ 
1: let  $s' := s$ 
2: if  $s.isCorrupted \equiv \text{FULLCORRUPT}$  then
3:   let  $s'.pendingRequests := \langle m, s.pendingRequests \rangle$  ▷ Collect incoming messages
4:   let  $m' \leftarrow d_{N^p}(s')$ 
5:   let  $a' \leftarrow \text{Addresses}$ 
6:   stop  $\{(a':a:m')\}, s'$ 
7: else if  $s.isCorrupted \equiv \text{CLOSECORRUPT}$  then
8:   let  $s'.pendingRequests := \langle m, s.pendingRequests \rangle$  ▷ Collect incoming messages
9:   let  $N^{\text{clean}} := N^p \setminus \{n | n \in^{\langle \rangle} s.nonces\}$ 
10:  let  $m' \leftarrow d_{N^{\text{clean}}}(s')$ 
11:  let  $a' \leftarrow \text{Addresses}$ 
12:  let  $s'.nonces := s.nonces$ 
13:  stop  $\{(a':a:m')\}, s'$ 
14: end if
15: let  $n, s' := \text{TAKENONCE}(s')$ 
16: if  $m \equiv \text{TRIGGER}$  then ▷ A special trigger message.
17:   let  $switch \leftarrow \{1, 2\}$ 
18:   if  $switch \equiv 1$  then ▷ Run some script.
19:     let  $\bar{w} \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}.documents \neq \langle \rangle$  if possible; otherwise stop  $\{\}, s'$ 
20:     let  $\bar{d} := \bar{w} +^{\langle \rangle} \text{activedocument}$ 
21:     RUNSCRIPT( $\bar{w}, \bar{d}, s'$ )
22:   else if  $switch \equiv 2$  then ▷ Create some new request.
23:     let  $w' := \langle n, \langle \rangle, \perp \rangle$ 
24:     let  $s'.windows := s'.windows +^{\langle \rangle} w'$ 
25:     let  $protocol \leftarrow \{P, S\}$ 
26:     let  $host \leftarrow \text{Domains}$ 
27:     let  $path \leftarrow \mathbb{S}$ 
28:     let  $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ 

```

```

29:     let  $n', s' := \text{TAKENONCE}(s')$ 
30:     let  $req := \langle \text{GET}, \text{host}, \text{HTTPReq}, n', \text{path}, \langle \rangle, \text{parameters}, \langle \rangle \rangle$ 
31:     SEND( $n, req, \text{protocol}, \perp, s'$ )
32:   end if
33: else if  $m \equiv \text{FULLCORRUPT}$  then ▷ Request to corrupt browser
34:   let  $s'.\text{isCorrupted} := \text{FULLCORRUPT}$ 
35:   stop  $\{\}, s'$ 
36: else if  $m \equiv \text{CLOSECORRUPT}$  then ▷ Close the browser
37:   let  $s'.\text{secrets} := \langle \rangle$ 
38:   let  $s'.\text{windows} := \langle \rangle$ 
39:   let  $s'.\text{pendingDNS} := \langle \rangle$ 
40:   let  $s'.\text{pendingRequests} := \langle \rangle$ 
41:   let  $s'.\text{sessionStorage} := \langle \rangle$ 
42:   let  $s'.\text{cookies} \subset \langle \rangle \text{Cookies}$  such that
      ( $c \in \langle \rangle s'.\text{cookies}$ )  $\iff$  ( $c \in \langle \rangle s.\text{cookies} \wedge c.\text{content.session} \equiv \perp$ )
43:   let  $s'.\text{isCorrupted} := \text{CLOSECORRUPT}$ 
44:   stop  $\{\}, s'$ 
45: else if  $\exists \langle \text{reference}, \text{request}, \text{key}, f \rangle \in \langle \rangle s'.\text{pendingRequests}$  such that
       $\pi_1(\text{dec}_s(m, \text{key})) \equiv \text{HTTPResp}$  then ▷ Encrypted HTTP response
46:   let  $m' := \text{dec}_s(m, \text{key})$ 
47:   if  $m'.\text{nonce} \neq \text{request}.\text{nonce}$  then
48:     stop  $\{\}, s$ 
49:   end if
50:   remove  $\langle \text{reference}, \text{request}, \text{key}, f \rangle$  from  $s'.\text{pendingRequests}$ 
51:   PROCESSRESPONSE( $m', \text{reference}, \text{request}, \text{S}, s'$ )
52: else if  $\pi_1(m) \equiv \text{HTTPResp} \wedge \exists \langle \text{reference}, \text{request}, \perp, f \rangle \in \langle \rangle s'.\text{pendingRequests}$  such that
       $m'.\text{nonce} \equiv \text{request}.\text{key}$  then
53:   remove  $\langle \text{reference}, \text{request}, \perp, f \rangle$  from  $s'.\text{pendingRequests}$ 
54:   PROCESSRESPONSE( $m, \text{reference}, \text{request}, \text{P}, s'$ )
55: else if  $m \in \text{DNSResponses}$  then ▷ Successful DNS response
56:   if  $m.\text{nonce} \notin s.\text{pendingDNS}$  then
57:     stop  $\{\}, s$ 
58:   end if
59:   let  $\langle \text{reference}, \text{message}, \text{protocol} \rangle := s.\text{pendingDNS}[m.\text{nonce}]$ 
60:   if  $\text{protocol} \equiv \text{S}$  then
61:     let  $k, s' := \text{TAKENONCE}(s')$ 
62:     let  $s'.\text{pendingRequests} := s'.\text{pendingRequests} + \langle \rangle \langle \text{reference}, \text{message}, k, m.\text{result} \rangle$ 
63:     let  $\text{message} := \text{enc}_a(\langle \text{message}, k \rangle, s'.\text{keyMapping}[\text{message}.\text{host}])$ 
64:   else
65:     let  $s'.\text{pendingRequests} := s'.\text{pendingRequests} + \langle \rangle \langle \text{reference}, \text{message}, \perp, m.\text{result} \rangle$ 
66:   end if
67:   let  $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$ 
68:   stop  $\{(m.\text{result}:a:\text{message})\}, s'$ 
69: else
70:   stop  $\{\}, s$ 
71: end if

```

## C DNS Servers

An informal description of a DNS server is given in Section 3.2. We now provide a formal definition of a DNS server  $d$ . A DNS server  $d$  is an atomic DY process  $(I^d, \{s_0^d\}, R^d, s_0^d, N^d)$  with

a finite set of addresses  $I^d$ . The (initial) state  $s_0^d \in [\mathbb{S} \times \text{Addresses}]$  is a mapping of a finite set of domain names to addresses. We note that our definition of DNS servers does not change the mapping from domain names to addresses at all. The set of states of the atomic DY process of a DNS server therefore contains only  $s_0^d$ .

We now specify the relation  $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$  of  $d$ . Just like in Appendix B.2, we describe this relation by a non-deterministic algorithm.

---

**Algorithm 9** Relation of a DNS server  $R^d$ 


---

**Input:**  $(a:f:m), s$

```

1: let  $domain, n$  such that  $\langle \text{DNSResolve}, domain, n \rangle \equiv m$  if possible; otherwise stop  $\{\}, s$ 
2: if  $domain \in s$  then
3:   let  $addr := s[domain]$ 
4:   let  $m' := \langle \text{DNSResolved}, addr, n \rangle$ 
5:   stop  $\{(f : a : m')\}, s$ 
6: end if
7: stop  $\{\}, s$ 

```

## D Step-By-Step Description of BrowserID (Primary IdP)

We now present additional details of the JavaScript implementation of BrowserID. While the basic steps have been shown in Section 4.2, we will now again refer to Figure 4 and provide a step-by-step description. As above, we focus on the main login flow without the CIF, and we leave out steps for fetching additional resources (like JavaScript files) and some less relevant postMessages and XHRs.

We (again) assume that the user uses a “fresh” browser, i.e., the user has not been logged in before. The user has already opened a document of some RP (RP-Doc) in her browser. RP-Doc includes a JavaScript file, which provides the BrowserID API. The user is now about to click on a login button in order to start a BrowserID login.

**Phase (i).** After the user has clicked on the login button, RP-Doc opens a new browser window, the *login dialog* (LD) [1]. The document of LD is loaded from LPO [2]. Now, LD sends a *ready* postMessage [3] to its opener, which is RP-Doc. RP-Doc then responds by sending a *request* postMessage [4]. This postMessage may contain additional information like a name or a logo of RP-Doc. LD then fetches the so-called *session context* from LPO using [5]. The session context contains information about whether the user is already logged in at LPO, which, by our assumption, is not the case at this point. The session context also contains an XSRF protection token which will be sent in all subsequent POST requests to LPO. Also, an `httpOnly` cookie called `browserid_state` is set, which contains an LPO session identifier. Now, the user is prompted to enter her email address (*login email address*), which she wants to use to log in at RP [6]. LD sends the login email address to LPO via an XHR [7], in order to get information about the IdP the email address belongs to. The information from this so-called *support document* may be cached at LPO for further use. LPO extracts the domain part of the login email address and fetches an information document [8] from a fixed path (`/.well-known/browserid`) at the IdP. This document contains the public key of IdP, and two paths, the provisioning path and the authentication path at IdP. These paths will be used later in the login process by LD. LPO converts these paths into URLs and sends them in its response [9] to the requesting XHR [7].

**Phase (ii).** As there is no record about the login email address in the localStorage under the

origin of LPO, the LD now tries to get a UC for this identity. For that to happen, the LD creates a new iframe, the *provisioning iframe* (PIF) [10]. The PIF's document is loaded [11] from the provisioning URL LD has just received before in [9]. The PIF now interacts with the LD via `postMessages` [12]. As the user is currently not logged in, the PIF tells the LD that the user is not authenticated yet. This also indicates to the LD that the PIF has finished operation. The LD then closes the PIF [13].

**Phase (iii).** Now, the LD saves the login email address in the `localStorage` indexed by a fresh nonce. This nonce is stored in the `sessionStorage` to retrieve the email address later from the `localStorage` again. Next, the LD navigates itself to the authentication URL it has received in [9]. The loaded document now interacts with the user and the IdP [14] in order to establish some authenticated session depending on the actual IdP implementation, which is out of scope of the BrowserID standard. For example, during this authentication procedure, the IdP may issue some session cookie.

**Phase (iv).** After the authentication to the IdP has been completed, the authentication document navigates the LD to the LD URL again. The LD's document is fetched again from LPO and the login process starts over. The following steps are similar to Phase (i): The ready and request `postMessages` are exchanged and the session context is fetched. As the user has not been authenticated to LPO yet, the session context still contains the same information as above in [5]. Now, the user is not prompted to enter her email address again. The email address is fetched from the `localStorage` under the index of the nonce stored in the `sessionStorage`. Now, the address information is requested again from LPO.

**Phase (v).** As there still is no UC belonging to the login email address in the `localStorage`, the PIF is created again. As the user now has established an authenticated session with the IdP, the PIF asks the LD to generate a fresh key pair. After the LD has generated the key pair [15], it stores the key pair in the `localStorage` (under the origin of LPO) and sends the public key to the PIF as a `postMessage` [16]. The following steps [17]–[19] are not specified in the BrowserID protocol. Typically, the PIF would send the public key to IdP (via an XHR) [17]. The IdP would create the UC [18] and send it back to the PIF [19]. The PIF then sends the UC to the LD [20], which stores it in the `localStorage`. Now, the LD closes the PIF.

**Phase (vi).** The LD is now able to create a CAP, as it has access to a UC and the corresponding private key in its `localStorage`. First, LD creates an IA for LPO [21]. The IA and the UC is then combined to a CAP, which is then sent to LPO in an XHR POST message [22]. LPO is now able to verify this CAP with the public key of IdP, which LPO has already fetched and cached before in [8]. If the CAP is valid, LPO considers its session with the user's browser to be authenticated for the email address the UC in the CAP is issued for.

**Phase (vii).** Now, in [23], the LD fetches a list of email addresses, which LPO considers to be owned by the user. If the login email address would not appear in this list, LD would abort the login process. After this, the LD fetches the address information about the login email address again in [24]. Using this information, LD validates if the UC is signed by the correct party (primary/secondary IdP). Now, LD generates an IA for the sender's origin of the request `postMessage` [4] (which was repeated in Phase (iv)) using the private key from the `localStorage` [25] (the IA is generated for the login email address). Also, it is recorded in the `localStorage` that the user is now logged in at RP with this email address. The LD then combines the IA with the UC stored in the `localStorage` to the CAP, which is then sent to RP-Doc in the *response* `postMessage` [26].

This concludes the login process that runs in LD. Afterwards, RP-Doc closes LD [27].

## E Sideshow/BigTent OpenID Flow

We will now give concrete examples of an OpenID flow that is started when Sideshow or BigTent want to authenticate a user via OpenID (as presented in Section 4.3). We will show typical requests and responses in such a flow and discuss the parameters that are important for the attacks presented in Section 5.2. We focus on Sideshow (and thus, Google), the URLs for BigTent (with Yahoo) are similar.

### E.1 OpenID Authentication Request

As we already discussed in Section 4.3, Sideshow maintains a session with the user. Sideshow issues UCs to a user only if the session is authenticated. This authentication is done via OpenID. When detecting that a session is not authenticated, Sideshow redirects the user's browser to the so-called *OpenID endpoint URL* of Google/Gmail. This URL may look as follows (line breaks added for readability):

```
https://www.google.com/accounts/o8/ud
?openid.mode=checkid_setup
&openid.ns=http%
&openid.ns.ax=http%
&openid.ax.mode=fetch_request
&openid.ax.type.email=http%
&openid.ax.required=email
&openid.ns.ui=http%
&openid.ui.mode=popup
&openid.identity=
  http%
&openid.claimed_id=
  http%
&openid.return_to=
  https%
&openid.realm=https%
```

In this URL, the parameter `openid.ax.type.email` encodes that Sideshow requests under the name “email” in the namespace “ax” an attribute of the type `http://axschema.org/contact/email`. Per definition of the OpenID attribute exchange schema,<sup>20</sup> this denotes the request for an email address. Note that Google's OpenID endpoint is (per the OpenID protocol) not obliged to follow this request and may issue an OpenID assertion without a (signed) email address.

The parameter `openid.return_to` contains the URL to which Google redirects the user after issuing the assertion. The assertion and possibly other information are appended to this URL.

### E.2 OpenID Authentication Response

After accessing the above OpenID endpoint URL, the user authenticates with Google and confirms that Google releases the requested information to Sideshow. Google then creates an

---

<sup>20</sup>See [http://openid.net/specs/openid-attribute-exchange-1\\_0.html](http://openid.net/specs/openid-attribute-exchange-1_0.html)

OpenID assertion and appends it to the `openid.return_to` URL that was contained in the OpenID authentication request. Finally, OpenID redirects the user's browser to the resulting URL, which may look as follows:

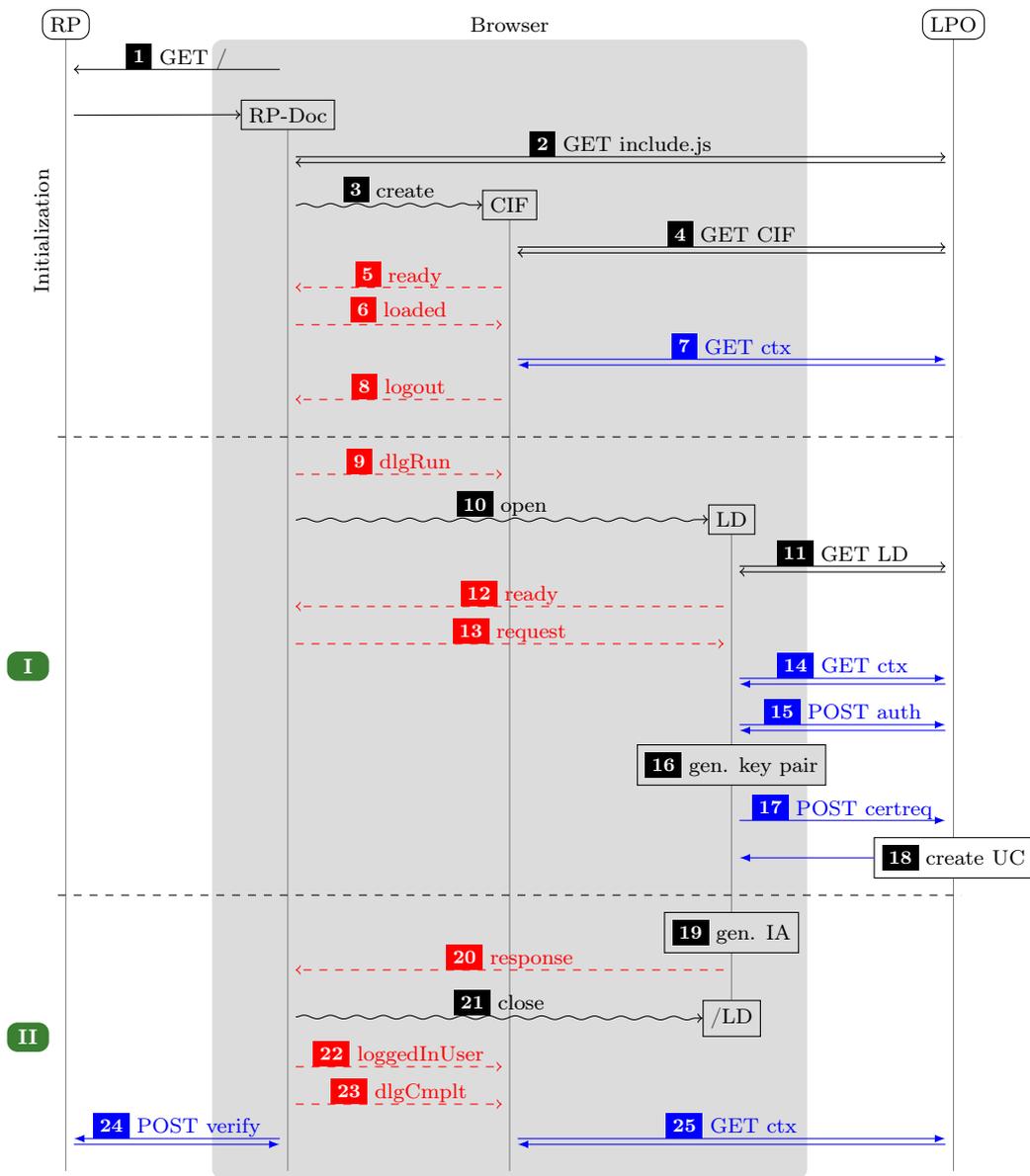
```
https://gmail.login.persona.org/authenticate/verify
?openid.ns=http%
&openid.mode=id_res
&openid.op_endpoint=https%
&openid.response_nonce=2013-09-24T11%
&openid.return_to=https%
&openid.assoc_handle=1.AM1YA(...)ubxC0qB
&openid.signed=op_endpoint
%
%
%
%
%
%
%
%
%
&openid.sig=BIPe1PIwitMp365MUEtd34IJLUs%
&openid.identity=
  https%
&openid.claimed_id=
  https%
&openid.ns.ext1=http%
&openid.ext1.mode=fetch_response
&openid.ext1.type.email=http%
&openid.ext1.value.email=user%
&openid.ns.ext2=http%
&openid.ext2.mode=popup
```

First note that the namespace “ax” was renamed by Gmail: What was prefixed with `openid.ax` in the request is now prefixed with `openid.ext1`. The parameter `openid.signed` contains the names of the parameters that have actually been MACed into the signature given in `openid.sig`. Note that the receiver of the assertion can not know the exact renaming performed by Gmail and must, although the renaming is obvious in this case, rely on the type parameters to determine the actual contents of the parameters. In this case, `openid.ext1.type.email` contains the AX schema type for an email address, saying that `openid.ext1.value.email` actually contains the requested email address.

The parameter `openid.assoc_handle` contains the ID of a temporary symmetric key created and stored at Google that is used for the MAC.

### E.3 Verification

After receiving the above request, Sideshow can forward all attributes to a verification service at Google. The URL for this verification is <https://www.google.com/accounts/o8/ud?openid>.



**Figure 6.** BrowserID sIdP flow overview. Black arrows (open tips) denote HTTP messages, blue arrows (filled tips) denote XHRs, red (dashed) arrows are postMessages, snake lines are commands to the browser.

mode=checkid\_authentication (same as in the authentication request, but with a different openid.mode value). Sideshow appends to this URL all attributes from the authentication response, i.e., the assertion, except for the openid.mode parameter. Google (only) checks that the MAC in openid.sig is correct (using the symmetric key stored earlier) and answers with “true” or “false” accordingly.

## F Step-By-Step Description of BrowserID (Secondary IdP)

We here provide a detailed description of the BrowserID flow including the LD and the CIF, but considering only the sIdP mode. The formal model and analysis of this setting is presented in Section 5 and Appendix G. As above, we leave out steps for fetching additional resources (like JavaScript files) and some less relevant postMessages.

As before, we assume that the user uses a “fresh” browser, i.e., the user has not been logged in before. Figure 6 shows a sequence diagram of the run. In comparison to the high-level description with Phases I and II in Section 4.1, a new phase, the *Initialization*, is shown, which contains some initialization steps and some steps of the CIF for automatic creation of CAPs.

**Initialization.** First, the user opens a web page of an RP (RP-Doc) in her browser [1]. RP-Doc includes the BrowserID JavaScript from LPO [2]. The JavaScript in RP-Doc then initializes the BrowserID JavaScript, which first creates a communication iframe (CIF) within RP-Doc [3]. The content of the CIF is loaded from LPO [4]. When the CIF has been initialized successfully, it sends a `ready` postMessage to the BrowserID JavaScript in RP-Doc [5], which in turn responds with the `loaded` postMessage [6]. This message may contain an email address, which we ignore for now (see below). The CIF saves the sender’s origin of this postMessage, as it identifies the RP it is working with.<sup>21</sup> It then fetches the session context from LPO using XHR [7]. The session context contains information about whether the user is already logged in at LPO, which, by our assumption, is not the case at this point. The session context also contains an XSRF protection token which will be sent in all subsequent POST requests to LPO. Also, an `httpOnly` cookie called `browserid_state` is set, which contains an LPO session identifier. The CIF finishes the initialization by sending a `logout` postMessage [8] to RP-Doc, indicating that the browser is currently not logged in at this RP. (In RP-Doc, this calls a message handler `onlogout` which RP may have registered.)

**Phase I.** Now the user starts to log in at RP, typically by clicking on a login button in RP-Doc, which calls the BrowserID login function. This JavaScript first tells the CIF that it will now open a login dialog window (LD) by sending the `dialog_running` postMessage [9]; this pauses the CIF, in particular, automatic CAP generation (see below). The BrowserID login function then opens the LD [10]. Its content is fetched from LPO [11]. When it is fully loaded, it sends a `ready` postMessage to the BrowserID JavaScript in RP-Doc [12], which is answered by sending a `request` postMessage back [13], indicating that the sender’s origin of this postMessage requests a CAP.

After this request, the dialog fetches the session context at LPO [14] (similar to what the CIF has done before). As the user is still not logged in, she is now asked to provide an email address and a password for LPO. These are then sent to LPO by an XHR [15]. If the credentials are accepted, a new key pair is generated by LD’s JavaScript [16] and the public key along with the entered email address is sent to LPO [17] in order to get a UC [18] (see Section 4.1). Moreover, the key pair and the UC are stored in the `localStorage` under the origin of LPO.

**Phase II.** Afterwards, an IA containing the so-called *audience* (the sender’s origin of the `request` postMessage [13]) and some expiration date is created, signed (with the generated private key), and combined with the certificate to a CAP [19]. In the `localStorage` it is recorded that the user is logged in at RP with the current email address. The CAP and the email address are now

---

<sup>21</sup>Note that for postMessages the sender origin cannot be spoofed and is always correct (see [19] for details).

sent back to RP-Doc in a **response** `postMessage` [20]. After this, the LD is closed [21].

The BrowserID JavaScript in RP-Doc informs the CIF that it now thinks that the email address received in the **response** `postMessage` is logged in [22]. Next, it tells the CIF that the LD is now closed [23], by which the CIF is awoken from pausing. The CIF then fetches the session context again [25] (as in [7]) in order to perform some additional checks (see below).

It is not specified in the BrowserID system how the RP-Doc has to process the CAP received in step [20]. Typically, as already mentioned in Section 4.1, the RP-Doc would send the CAP to the RP's server [24], which then can verify the CAP. If successful, RP can consider the user (with the email address mentioned in the CAP) to be logged in and send her some token, the RP service token (as introduced in Section 4.1).

### F.1 Additional Checks

We note that when `postMessages` are sent, the BrowserID system makes certain checks. These checks are carried out by two different (Mozilla) JavaScript libraries. The communication between RP-Doc and CIF is realized with the library *JSChannel* and the communication between RP-Doc and LD is realized with the library *WinChan*.

First recall that `postMessages` can be sent by providing the receiver's origin. The browser ensures that such a `postMessage` can only be received by a document having the origin the sender has specified. If the receiver's origin does not match the one specified by the sender, the sender receives a JavaScript exception. However, the sender is not required to provide a receiver's origin, so any receiver can receive the `postMessage`. Also, a receiver can check from which window a `postMessage` was sent and which origin the sender belongs to.

Now, the CIF only sends or accepts `postMessages` to or from its parent window (which typically should be RP-Doc). However, the CIF does not check any origin while receiving `postMessages` and does not provide an origin when sending `postMessages`. When RP-Doc receives a message it expects to come from CIF, RP-Doc checks if the origin of this message matches LPO and if the sender's window is the window of CIF. RP-Doc always provides LPO as the receiver's origin when sending messages to the window it believes to contain the CIF.

The LD sends [12] only to its opener (RP-Doc) without providing any receiver's origin to check. After this, the LD accepts only one **request** `postMessage` [13] and blocks any further incoming `postMessages`. The sender's origin of the **request** `postMessage` [13] is used by LD to determine the receiver's origin of the **response** `postMessage` [20]. LD also fixes the receiver of [20] to be its opener. When RP-Doc sends the **request** `postMessage` [13] to the LD, it sets the receiver's origin to be LPO in the `postMessage`. However, any `postMessage` RP-Doc expects to be sent by LD is not checked (see also Section 5.2).

During the interaction between RP-Doc and LD, an additional check is set in place at both parties: If one of both documents is navigated away, the window of LD is closed immediately (and therefore any process in the LD is aborted).

We also note that step [23] triggers two checks in the CIF: First, the CIF checks the current login status at LPO, by fetching the session context [25]. Second, the CIF compares the email address received in [22] to the one that is marked as being logged in at RP in the `localStorage` (under the origin of LPO). If in one of the checks the user is considered to be not logged in, a **logout** `postMessage` is sent to RP-Doc (similarly to [8]). Otherwise, if in the second check a mismatch is detected, the CIF creates a new CAP according to the information in the `localStorage` and sends it as a so-called **login** `postMessage` to RP-Doc. Whether this CAP is used by RP-Doc or the one received in step [20] depends on the way the RP-Doc uses the API provided by BrowserID. One

possibility (which is considered in the BrowserID test implementation) is that RP-Doc relays all received CAPs to the RP server with an XHR. The RP server, as already mentioned above, would then verify each CAP it receives and issue an RP service token every time. This is also what is done in our model of the BrowserID system.

## F.2 Automatic CAP Creation

Once a run as described above is completed, an RP-Doc can get CAPs directly during the Initialization from the CIF: The CIF will automatically issue a fresh CAP and send it to RP-Doc (in a `login` postMessage instead of [8]) iff (1) some email address is marked as logged in at RP in the localStorage (under the origin LPO), (2) if an email address was provided in the `loaded` postMessage [6], this email address differs from the one recorded in the localStorage, and (3) the user is logged in at LPO (indicated in [7]). If necessary, a new key pair is created and a corresponding new UC is requested at LPO.

## F.3 LPO Session

As mentioned before, in the initial run LPO establishes a session with the browser by setting a cookie `browserid_state` (in step [7]) on the client-side.

If such a run is started again (possibly with some other RP) with the same browser in an LPO session in which the user is already logged in at LPO, the user is not asked again by the LD to provide her credentials. Instead she is presented a list of her email addresses (which is fetched from LPO and cached in the localStorage) in order to choose one address. Then, she is asked if she trusts the computer she is using and is given the option to be logged in for one month or “for this session only” (*ephemeral* session). However, in any case cookies will be stored for some time in the browser and will be valid for some time on the LPO server (one hour to 30 days).

## F.4 Logout

We have to differentiate between two ways of logging out: an RP logout and an LPO logout. An RP logout is handled by the CIF after it has received a `logout` postMessage from RP-Doc. The CIF changes the localStorage (under the origin of LPO) such that no email address is recorded to be logged in at RP and replies to RP-Doc with a `logout` postMessage. RP-Doc can run some callback it may have registered before.

An LPO logout essentially requires to logout at the web site of LPO. The LPO logout removes all key pairs and certificates from the localStorage and invalidates the session on the LPO server.

# G BrowserID Model

In this section, we provide the full BrowserID model, i.e., the web system  $\mathcal{BID} = (\mathcal{W}, \mathcal{S}, \text{script}, E_0)$ , and its security properties. We note that our model considers the BrowserID system with the fixes proposed in Section 5.2.

We first note that a UC  $uc$  for the identity (email address)  $i$  and public key (verification key)  $\text{pub}(k_u)$ , where  $k_u$  is the private key (signing key) of the user, is modeled to be a message of the form  $uc = \text{sig}(\langle i, \text{pub}(k_u) \rangle, k^{\text{LPO}})$ , where  $k^{\text{LPO}}$  is the signing key of LPO. An IA  $ia$  for an origin  $o = \langle \text{example.com}, \mathcal{S} \rangle$  is a message of the form  $ia = \text{sig}(o, k'_u)$ . Now, a CAP is of the form  $\langle uc, ia \rangle$ . We call that CAP *valid* iff  $k_u = k'_u$ . Note that the time stamps are omitted both

from the UC and the IA. This models that both certificates are valid indefinitely. In reality, as explained in Section 4, they are valid for a certain period of time, as indicated by the time stamps. So our modeling is a safe overapproximation.

The set  $\mathcal{N}$  of nonces is partitioned into three disjoint sets, an infinite set  $N^{\mathcal{W}}$ , an infinite set  $K_{\text{private}}$ , and a finite set **Secrets**. The set  $N^{\mathcal{W}}$  is further partitioned into infinite sets of nonces, one set  $N^p \subseteq N^{\mathcal{W}}$  for every  $p \in \mathcal{W}$ .

The sets **Addresses** and **Domains** have already been introduced in Section 5.3. Let **addr** and **dom** denote the assignments from atomic processes to sets of **Addresses** and **Domains**, respectively, where browsers (in **B**) do not have a domain. If **dom** or **addr** returns a set with only one element, we often write **dom**( $x$ ) or **addr**( $x$ ) to refer to the element. Let **key**: **Domains**  $\rightarrow$   $K_{\text{private}}$  be an injective mapping that assigns a private key to every domain. The set **Secrets**  $\subseteq \mathcal{N}$  is the set of passwords (secrets) the browsers share with LPO. Let **ID** be the finite set of an email address (IDs) of the form  $\langle \text{name}, d \rangle$ , with  $\text{name} \in \mathbb{S}$  and  $d \in \text{Domains}$ , registered at LPO. As mentioned in Section 5.3, different browsers own disjoint sets of secrets and different secrets are assigned disjoint sets of IDs. Let **ownerOfSecret** : **Secrets**  $\rightarrow$  **B** denote the mapping that assigns to each secret the browser that owns this secret. Let **secretOfID** : **ID**  $\rightarrow$  **Secrets** denote the mapping that assigns to each identity the secret it belongs to. Now, we define the mapping **ownerOfID** : **ID**  $\rightarrow$  **B**,  $i \mapsto \text{ownerOfSecret}(\text{secretOfID}(i))$ , which assigns to each identity the browser that owns this identity.

We are now ready to define the attacker, the browsers, LPO, the relying parties, and the scripts used in BrowserID in more detail. As for the attacker and browsers, nothing much needs to be specified. The attacker will be modeled as a network attacker as specified in Section 3.1 and browsers will simply be web browsers as specified in Section 3.4. We only need to fix the addresses the attacker and the browsers can listen to and their initial states.

### G.1 Attacker

As mentioned, the attacker **attacker** is modeled to be a network attacker as specified in Section 3.1. We allow it to listen to/spoof all available IP addresses, and hence, define  $I^{\text{attacker}} = \text{Addresses}$ . His initial state is  $s_0^{\text{attacker}} = \langle \text{attdoms}, \text{pubkeys} \rangle$ , where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker and *pubkeys* is a sequence of all domains and the corresponding public keys. All other parties use the attacker as a DNS server.

### G.2 Browsers

Each  $b \in \mathbf{B}$  is a web browser as defined in Section 3.4, with  $I^b := \{\text{addr}(b)\}$  being its address and the initial state  $s_0^b$  defined as follows: the key mapping maps every domain to its public key, according to the mapping **key**; the DNS address is **addr**(attacker); the secrets are those owned by  $b$  (as defined above) and they are indexed by the origin  $\langle \text{dom}(\text{LPO}), \mathbb{S} \rangle$ ; *sts* is  $\langle \text{dom}(\text{LPO}) \rangle$ . (Without the latter, the attacker could trivially inject, by an HTTP response, its own **browserid\_state** cookie and by this violate property **(B)**.)

### G.3 LPO

LPO is an atomic DY process  $(I^{\text{LPO}}, Z^{\text{LPO}}, R^{\text{LPO}}, s_0^{\text{LPO}}, N^{\text{LPO}})$  with the IP address  $I^{\text{LPO}} = \{\text{addr}(\text{LPO})\}$ . The initial state  $s_0^{\text{LPO}}$  of LPO contains the private key of its domain, its signing key  $k^{\text{LPO}}$ , all secrets in **Secrets** and the corresponding sequences of IDs. (LPO does not need the

public keys of other parties, which is why we omit them from LPO's initial state.). The definition of  $R^{\text{LPO}}$  follows the description of LPO in Section 4 in a straightforward way. HTTP responses by LPO can contain, besides complex terms (e.g., for XHR responses), strings representing scripts, namely the script `script_LPO_cif` run in the CIF and the script `script_LPO_ld` run in the LD. These scripts are defined in Appendices G.5.2 and G.5.3, respectively.

Before we provide a detailed formal specification of LPO, we first provide an informal description.

**Client sessions at LPO.** Any party can establish a *session* at LPO. Such a session can either be authenticated or unauthenticated. Roughly speaking, a session becomes authenticated if a client has provided a secret  $sec \in \text{Secrets}$  to LPO in the session. Such a session is then authenticated for all IDs associated with  $sec$ , i.e., for all such IDs an UC can be requested from LPO within the authenticated session. (Recall that for every secret, LPO contains a list of all IDs associated with that secret.) An authenticated session can (non-deterministically) *expire*, i.e. the authenticated session can get unauthenticated or it is removed completely. Such an expiration is used to model a user logout or a session expiration caused by a timeout.

More specifically, a session is identified by a nonce, which is issued by LPO. Moreover, a session is associated with some `xsrftoken`, which is also a nonce issued by LPO. LPO stores all information about established sessions in its state as a dictionary indexed by the session identifier. In this dictionary, for every session LPO stores a pair containing the `xsrftoken` and, in authenticated sessions, the sequence of all IDs associated with the secret provided in the session, or, in unauthenticated sessions, the empty sequence  $\langle \rangle$  of IDs. On the receiver side (typically a browser) LPO placed, by appropriate headers in its HTTPS responses, a cookie named `browserid_state` whose value is the session identifier (a nonce). This cookie is flagged to be a session, `httpOnly`, and secure cookie.

**HTTPSRequests to LPO.** LPO answers only to certain requests (listed below). All such requests have to be over HTTPS. Also, all responses send by LPO contain the `Strict-Transport-Security` header.

GET `/cif`. LPO replies to this request by providing the script `script_LPO_cif`.

GET `/ld`. LPO replies to this request by providing the script `script_LPO_ld`.

GET `/ctx`. LPO replies with a session context. More precisely, LPO first checks if a cookie `browserid_state` was sent within this request and if its value identifies a session within LPO's state. If such a session exists, LPO responds to such a request with the list of (authenticated) IDs for this session,<sup>22</sup> the `xsrftoken`, and a `Set-Cookie` header, which sets the `browserid_state` cookie. If no cookie `browserid_state` was sent in the request, or if the value of the cookie `browserid_state` does not identify a session within LPO's state, LPO first creates a new session. Such a new session contains a fresh nonce as a session identifier, the empty sequence  $\langle \rangle$  of IDs, and a fresh nonce as a `xsrftoken`. Once such a session is created, LPO responds as above.

POST `/auth`. This request is sent to authenticate a session at LPO. A request to this interface has to contain some secret  $sec \in \text{Secrets}$  in its body. The request also has to contain the cookie `browserid_state` which has to refer to some session in the state of LPO. Moreover,

---

<sup>22</sup>In the real implementation, the session context only contains a flag indicating the authentication state of the session. However, another GET request interface is available to retrieve the list of authenticated IDs for the current session. Here, for simplicity, we right away provide all authenticated IDs in the session context.

the request has to contain an `xsrftoken` in its body which has to match the one recorded in the considered session in LPO's state. The session recorded in the state of LPO is then modified to include the sequence of all IDs associated with `sec`. The response to such a request contains some static acknowledgment.

**POST /certreq.** Such a request is sent to LPO in order to request a UC. The request has to contain an ID and a public key in its body, for which a UC is requested. The request also has to contain a cookie named `browserid_state` which has to refer to some session recorded in the state of LPO. Moreover, the request has to contain an `xsrftoken` in its body which matches the `xsrftoken` in the considered session record in LPO's state. Also, the sequence of IDs in the considered session recorded in LPO's state has to contain the ID provided in the (body of the) request. This ID is then paired with the public key sent in the request and the resulting pair is then signed with  $k^{\text{LPO}}$ . In other words, a UC is created for the ID and the public key provided in the request. Finally, LPO responds with this UC.

We now define LPO formally as an atomic DY process  $(I^{\text{LPO}}, Z^{\text{LPO}}, R^{\text{LPO}}, s_0^{\text{LPO}}, N^{\text{LPO}})$ . As already mentioned, we define  $I^{\text{LPO}} = \{\text{addr}(\text{LPO})\}$ . In order to define the set  $Z^{\text{LPO}}$  of states of LPO, we first define the terms describing the session context of a session.

**Definition 25.** A term of the form  $\langle \text{ids}, \text{xsrftoken} \rangle$  with  $\text{ids} \subset^{\diamond} \text{ID}$  and  $\text{xsrftoken} \in \mathcal{N}$  is called an LPO session context. We denote the set of all LPO session contexts by `LPOSessionCTXs`.

Now, we define the set  $Z^{\text{LPO}}$  of states of LPO as well as the initial state  $s_0^{\text{LPO}}$  of LPO.

**Definition 26.** A state  $s \in Z^{\text{LPO}}$  of LPO is a term of the form  $\langle \text{nonces}, \text{sslkey}, \text{signkey}, \text{sessions}, \text{secrets} \rangle$  where  $\text{nonces} \subset^{\diamond} \mathcal{N}$  (used nonces),  $\text{sslkey} = \text{key}(\text{dom}(\text{LPO}))$ ,  $\text{signkey} = k^{\text{LPO}}$ ,  $\text{sessions} \in [\mathcal{N} \times \text{LPOSessionCTXs}]$ , and  $\text{secrets} \in [\text{Secrets} \times \mathcal{T}_{\mathcal{N}}]$  is a dictionary which assigns to every secret  $\text{sec} \in \text{Secrets}$  the sequence of all IDs associated with  $\text{sec}$ .<sup>23</sup>

The initial state  $s_0^{\text{LPO}}$  of LPO is a state of LPO with  $s_0^{\text{LPO}}.\text{nonces} = \langle \rangle$  and  $s_0^{\text{LPO}}.\text{sessions} = \langle \rangle$ .

**Example 6.** A possible state  $s$  of LPO may look like this:

$$s = \langle \langle n_1, \dots, n_m \rangle, k, k^{\text{LPO}}, \text{sessions}, \text{secrets} \rangle$$

with

$$\begin{aligned} \text{sessions} &= \langle \langle \text{sessionid}_1, \langle \langle id'_1, \dots, id'_l \rangle, \text{xsrftoken} \rangle \rangle, \dots \rangle \\ \text{secrets} &= \langle \langle \text{secret}_1, \langle id_1, \dots, id_p \rangle \rangle, \dots \rangle \end{aligned}$$

We now specify the relation  $R^{\text{LPO}} \subseteq (\mathcal{E} \times Z^{\text{LPO}}) \times (2^{\mathcal{E}} \times Z^{\text{LPO}})$  of LPO. Just like in Appendix B.2, we describe this relation by a non-deterministic algorithm. We note that we use the function `TAKENONCE` introduced in Section B.2.1 for this purpose.

---

<sup>23</sup>As mentioned before, the state of LPO does not need to contain public keys.

**Algorithm 10** Relation of LPO  $R^{\text{LPO}}$ 


---

**Input:**  $(a:f:m), s$

- 1: **let**  $s' := s$
- 2: **if**  $m \equiv \text{TRIGGER}$  **then** ▷ Triggers a (non-deterministic) logout or session expiration
- 3:   **if**  $s'.\text{sessions} \neq \langle \rangle$  **then**
- 4:     **let**  $\text{sessionid} \leftarrow \{id_j | id_j \in s'.\text{sessions}\}$
- 5:     **if**  $\text{sessionid} \in \langle \rangle s'.\text{sessions}$  **then**
- 6:       **let**  $\text{choice} \leftarrow \{\text{logout}, \text{expire}\}$
- 7:       **if**  $\text{choice} \equiv \text{logout}$  **then**
- 8:          **let**  $\text{session} := s'.\text{sessions}[\text{sessionid}]$
- 9:          **let**  $\text{session}[\text{ids}] := \langle \rangle$
- 10:         **let**  $s'.\text{sessions}[\text{sessionid}] := \text{session}$
- 11:         **stop**  $\{\}, s'$
- 12:     **else**
- 13:       **remove** the element with key  $\text{sessionid}$  from the dictionary  $s'.\text{sessions}$
- 14:       **stop**  $\{\}, s'$
- 15:     **end if**
- 16:   **end if**
- 17: **end if**
- 18: **else**
- 19:   **let**  $m_{\text{dec}}, k'$  **such that**  $\langle m_{\text{dec}}, k' \rangle \equiv \text{dec}_a(m, s'.\text{sslkey})$  **if possible; otherwise stop**  $\{\}, s$
- 20:   **let**  $n, \text{method}, \text{path}, \text{params}, \text{headers}, \text{body}$  **such that**  $\langle \text{HTTPReq}, n, \text{method}, \text{dom}(\text{LPO}), \text{path}, \text{params}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$  **if possible; otherwise stop**  $\{\}, s$
- 21:   **if**  $\text{method} \equiv \text{GET} \wedge \text{path} \equiv /cif$  **then** ▷ Deliver CIF script
- 22:     **let**  $m' :=$   
 $\text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle, \langle \text{script\_LPO\_cif}, \text{initState}_{\text{cif}} \rangle \rangle, k')$   
 where  $\text{initState}_{\text{cif}}$  is the initial scriptstate of  $\text{script\_LPO\_cif}$  according to Definition 28.
- 23:     **let**  $E := \{(f:a:m')\}$
- 24:     **stop**  $E, s'$
- 25:   **else if**  $\text{method} \equiv \text{GET} \wedge \text{path} \equiv /ld$  **then** ▷ Deliver LD script.
- 26:     **let**  $m' :=$   
 $\text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle, \langle \text{script\_LPO\_ld}, \text{initState}_{\text{ld}} \rangle \rangle, k')$   
 where  $\text{initState}_{\text{ld}}$  is the initial scriptstate of  $\text{script\_LPO\_ld}$  according to Definition 29.
- 27:     **let**  $E := \{(f:a:m')\}$
- 28:     **stop**  $E, s'$
- 29:   **else if**  $\text{method} \equiv \text{GET} \wedge \text{path} \equiv /ctx$  **then** ▷ Deliver context information.
- 30:     **let**  $\text{sessionid} := \text{headers}[\text{Cookie}][\text{browserid\_state}]$
- 31:     **if**  $\text{sessionid} \notin \langle \rangle s'.\text{sessions}$  **then** ▷ Create new session if needed.
- 32:       **let**  $\text{sessionid}, s' := \text{TAKENONCE}(s')$
- 33:       **let**  $\text{ids} := \langle \rangle$
- 34:       **let**  $\text{xsrftoken}, s' := \text{TAKENONCE}(s')$
- 35:       **let**  $s'.\text{sessions} := s'.\text{sessions} + \langle \rangle \langle \text{sessionid}, \langle \text{ids}, \text{xsrftoken} \rangle \rangle$
- 36:     **end if**
- 37:     **let**  $\text{result} := s'.\text{sessions}[\text{sessionid}]$
- 38:     **let**  $\text{headers}' := \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle,$   
 $\langle \text{Set-Cookie}, \langle \langle \text{browserid\_state}, \langle \text{sessionid}, \top, \top, \top \rangle \rangle \rangle \rangle$
- 39:     **let**  $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \text{headers}', \text{result} \rangle, k')$
- 40:     **let**  $E := \{(f:a:m')\}$
- 41:     **stop**  $E, s'$
- 42:   **else if**  $\text{method} \equiv \text{POST} \wedge \text{path} \equiv /auth$  **then** ▷ Authenticate session.
- 43:     **let**  $\text{sessionid} := \text{headers}[\text{Cookie}][\text{browserid\_state}]$
- 44:     **let**  $\text{secret}, \text{xsrftoken}$  **such that**  $\langle \text{secret}, \text{xsrftoken} \rangle \equiv \text{body}$  **if possible; otherwise stop**  $\{\}, s$

```

45:   if  $sessionid \in \langle \rangle s'.sessions$  then
46:     if  $secret \in \langle \rangle s'.secrets \wedge s'.sessions[sessionid].xsrftoken \equiv xsrftoken$  then
47:       let  $ids := s'.secrets[secret]$ 
48:       let  $s'.sessions[sessionid].ids := ids$ 
49:       let  $m' := enc_s(\langle \text{HTTPResp}, n, 200, \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle, \top \rangle, k')$ 
50:       let  $E := \{(f:a:m')\}$ 
51:       stop  $E, s'$ 
52:     end if
53:   end if
54: else if  $method \equiv \text{POST} \wedge path \equiv /certreq$  then ▷ Sign pubkey, deliver UC
55:   let  $sessionid := headers[\text{Cookie}][browserid\_state]$ 
56:   let  $ids := s'.sessions[sessionid].ids$ 
57:   let  $xsrftoken := s'.sessions[sessionid].xsrftoken$ 
58:   let  $id, pubkey, xsrftoken'$  such that  $\langle id, pubkey, xsrftoken' \rangle \equiv body$  if possible; otherwise
stop  $\{\}, s$ 
59:   if  $id \in \langle \rangle ids \wedge xsrftoken \equiv xsrftoken'$  then
60:     let  $uc := sig(\langle id, pubkey \rangle, s'.signkey)$ 
61:     let  $m' := enc_s(\langle \text{HTTPResp}, n, 200, \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle, uc \rangle, k')$ 
62:     let  $E := \{(f:a:m')\}$ 
63:     stop  $E, s'$ 
64:   end if
65: end if
66: end if
67: stop  $\{\}, s$ 

```

## G.4 Relying Parties

A relying party  $r \in \text{RP}$  is a web server modeled as an atomic DY process  $(I^r, Z^r, R^r, s_0^r, N^r)$  with the address  $I^r := \{\text{addr}(r)\}$ . Its initial state  $s_0^r$  contains its domain, the private key associated with its domain, the public key of LPO, and the set of service token it has provided. The definition of  $R^r$  again follows the description in Appendix F. RP only accepts messages sent over HTTPS. Whenever  $r$  receives a GET message, it returns the script `script_RP_index` (see below) and sets the `Strict-Transport-Security` header. If  $r$  receives an HTTPS POST message, it checks if (1) the message contains a valid CAP for  $r$ , and (2) the header of the message contains an `Origin` header which only contains a single origin and this origin is  $r$ 's domain with HTTPS. If this check is successful,  $r$  responds with a token of the form  $\langle n, i \rangle$  (sent in the body of the response), where  $i \in \text{ID}$  is the ID for which the CAP was issued and  $n$  is a freshly chosen nonce. We call, as mentioned in Section 5.3,  $\langle n, i \rangle$  an *RP service token (for ID  $i$ )*. As mentioned,  $r$  keeps a list of such tokens in its state. Intuitively, a client in possession of such a token can use the service of  $r$  for ID  $i$  (e.g., access data of  $i$  at  $r$ ).

We now provide the formal definition of  $r$  as an atomic DY process  $(I^r, Z^r, R^r, s_0^r, N^r)$ . As mentioned, we define  $I^r = \{\text{addr}(r)\}$ . Next, we define the set  $Z^r$  of states of  $r$  and the initial state  $s_0^r$  of  $r$ .

**Definition 27.** A state  $s \in Z^r$  of an RP  $r$  is a term of the form  $\langle \text{nonces}, \text{sslkey}, \text{domain}, \text{pubkLPO}, \text{serviceTokens} \rangle$  where  $\text{nonces} \subset \langle \rangle \mathcal{N}$  (used nonces),  $\text{sslkey} = \text{key}(\text{dom}(\text{LPO}))$ ,  $\text{domain} = \text{dom}(r)$ ,  $\text{pubkLPO} = \text{pub}(\text{key}(\text{dom}(\text{LPO})))$ ,  $\text{serviceTokens} \in [\mathcal{N} \times \mathbb{S}]$ .

The initial state  $s_0^r$  of  $r$  is a state of  $r$  with  $s_0^r.\text{nonces} = \langle \rangle$  and  $s_0^r.\text{serviceTokens} = \langle \rangle$ .

We now specify the relation  $R^r \subseteq (\mathcal{E} \times Z^r) \times (2^{\mathcal{E}} \times Z^r)$  of  $r$ . Just like in Appendix B.2,

we describe this relation by a non-deterministic algorithm. We note that we use the function TAKENONCE introduced in Section B.2.1 for this purpose.

---

**Algorithm 11** Relation of a Relying Party  $R^r$ 


---

**Input:**  $(a:f:m), s$

- 1: **let**  $s' := s$
- 2: **let**  $m_{\text{dec}}, k'$  **such that**  $\langle m_{\text{dec}}, k' \rangle \equiv \text{dec}_a(m, s'.\text{sslkey})$  **if possible; otherwise stop**  $\{\}, s$
- 3: **let**  $n, \text{method}, \text{path}, \text{params}, \text{headers}, \text{body}$  **such that**  $\langle \text{HTTPReq}, n, \text{method}, s'.\text{domain}, \text{path}, \text{params}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$  **if possible; otherwise stop**  $\{\}, s$
- 4: **if**  $\text{method} \equiv \text{GET}$  **then**  $\triangleright$  Deliver RP's index script
- 5:   **let**  $m' :=$   
     $\text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \langle \text{Strict-Transport-Security}, \top \rangle \rangle, \langle \text{script\_RP\_index}, \text{initState}_{\text{rp\_index}} \rangle \rangle, k')$   
    where  $\text{initState}_{\text{rp\_index}}$  is the initial scriptstate of  $\text{script\_RP\_index}$  according to Definition 30.
- 6:   **let**  $E := \{(f:a:m')\}$
- 7:   **stop**  $E, s'$
- 8: **else if**  $(\text{method} \equiv \text{POST}) \wedge (\text{headers} \equiv \langle \langle \text{Origin}, \langle s'.\text{domain}, \mathcal{S} \rangle \rangle \rangle)$  **then**  $\triangleright$  Check received CAP
- 9:   **let**  $uc, ia$  **such that**  $\langle uc, ia \rangle \equiv \text{body}$  **if possible; otherwise stop**  $\{\}, s$   $\triangleright$  Extract UC and IA
- 10:   **let**  $i := \pi_1(\text{extractmsg}(uc))$   $\triangleright$  Extract ID from UC
- 11:   **let**  $pku := \pi_2(\text{extractmsg}(uc))$   $\triangleright$  Extract pubkey from UC
- 12:   **let**  $o := \text{extractmsg}(ia)$   $\triangleright$  Extract audience from IA
- 13:   **if**  $(\text{checksig}(uc, s'.\text{pubkLPO}) \equiv \top) \wedge (\text{checksig}(ia, pku) \equiv \top \wedge o \equiv \langle s'.\text{domain}, \mathcal{S} \rangle)$  **then**
- 14:     **let**  $n', s' := \text{TAKENONCE}(s')$   $\triangleright$  Issue service token
- 15:     **let**  $s'.\text{serviceTokens} := s'.\text{serviceTokens} + \langle n', i \rangle$
- 16:     **let**  $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, \langle n', i \rangle \rangle, k')$
- 17:     **let**  $E := \{(f:a:m')\}$
- 18:     **stop**  $E, s'$
- 19:   **end if**
- 20: **end if**
- 21: **stop**  $\{\}, s$

## G.5 BrowserID Scripts

As already mentioned in Section 5.3, the set  $\mathcal{S}$  of the web system  $\mathcal{BID} = (\mathcal{W}, \mathcal{S}, \text{script}, E_0)$  consists of the scripts  $R^{\text{att}}$ ,  $\text{script\_RP\_index}$ ,  $\text{script\_LPO\_cif}$ , and  $\text{script\_LPO\_ld}$  with their string representations  $\text{att\_script}$ ,  $\text{script\_RP\_index}$ ,  $\text{script\_LPO\_cif}$ , and  $\text{script\_LPO\_ld}$  (defined by  $\text{script}$ ). The script  $R^{\text{att}}$  is the attacker script (see Section 3.1). The formal model of the latter two scripts follows the description in Appendix F in a straightforward way. The script  $\text{script\_RP\_index}$  defines the script of the RP index page. In reality, this page has its own script(s) and includes a script from LPO. In our model, we combine both scripts to  $\text{script\_RP\_index}$ . In particular, this script is responsible for creating the CIF and the LD iframes/subwindows, whose content (scripts) are loaded from LPO.

In what follows, the scripts  $\text{script\_RP\_index}$ ,  $\text{script\_LPO\_cif}$ , and  $\text{script\_LPO\_ld}$  are defined formally. First, we introduce some notation and helper functions.

### G.5.1 Some Notation and Helper Functions

In the formal description of the scripts we use an abbreviation for URLs for LPO. We write  $\text{URL}_{\text{path}}^{\text{LPO}}$  to describe the following URL term:  $\langle \text{URL}, \mathcal{S}, \text{dom}(\text{LPO}), \text{path}, \langle \rangle \rangle$ . Also, we call  $\text{origin}_{\text{LPO}}$  the origin of LPO which describes the following origin term:  $\langle \text{dom}(\text{LPO}), \mathcal{S} \rangle$ .

The localStorage under the origin of LPO used by the scripts  $\text{script\_LPO\_cif}$  and

*script\_LPO\_ld* is organized as follows: it is a dictionary containing only one entry. This entry consists of the key `siteInfo` and (as its value) a dictionary where this dictionary has origins as keys with IDs as values indicating that a certain ID (of the user) is logged in at the referenced origin. Here is an example a possible `localStorage`.

**Example 7.**

$$\langle\langle\text{siteInfo}, \langle\langle\langle\text{domain}_{\text{RP1}}, \text{S}\rangle, \text{id}_1\rangle, \langle\langle\text{domain}_{\text{RP2}}, \text{S}\rangle, \text{id}_1\rangle, \langle\langle\text{domain}_{\text{RP3}}, \text{S}\rangle, \text{id}_2\rangle\rangle\rangle\rangle \quad (16)$$

This example shows a `localStorage` under the origin of `LPO`, indicating that the user is logged in at `domainRP1` and `domainRP2` with `id1` and at `domainRP3` with `id2` (using `HTTPS`).

In order to simplify the description of the scripts, several helper functions are used.

**CHOOSEINPUT.** As explained in Section 3.4, the state of a document contains a term, say *scriptinputs*, which records the input this document has obtained so far (via XHRs and `postMessages`). If the script of the document is activated, it will typically need to pick one input message from *scriptinputs* and record which input it has already processed. For this purpose, the function `CHOOSEINPUT(s', scriptinputs)` is used, where *s'* denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate *s'* and chooses a yet unhandled input message from *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

---

**Algorithm 12** Choose an unhandled input message for a script

---

```

1: function CHOOSEINPUT(s', scriptinputs)
2:   let iid such that  $iid \in \{1, \dots, |\text{scriptinputs}|\} \wedge iid \notin \langle \rangle s'.\text{handledInputs}$  if possible; otherwise
   return ( $\perp$ , s')
3:   let input :=  $\pi_{iid}(\text{scriptinputs})$ 
4:   let s'.handledInputs := s'.handledInputs +  $\langle \rangle iid$ 
5:   return (input, s')
6: end function

```

**PARENTWINDOW.** To determine the nonce referencing the parent window in the browser, the function `PARENTWINDOW(tree, docnonce)` is used. It takes the term *tree*, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce *docnonce*, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the window which directly contains in its subwindows the window of the document referenced by *docnonce*. If there is no such window (which is the case if the script runs in a document of a top-level window), `PARENTWINDOW` returns  $\perp$ .

**SUBWINDOWS.** This function takes a term *tree* and a document nonce *docnonce* as input just as the function above. If *docnonce* is not a reference to a document contained in *tree*, then `SUBWINDOWS(tree, docnonce)` returns  $\langle \rangle$ . Otherwise, let  $\langle \text{docnonce}, \text{origin}, \text{script}, \text{scriptstate}, \text{scriptinput}, \text{subwindows}, \text{active} \rangle$  denote the subterm of *tree* corresponding to the document referred to by *docnonce*. Then, `SUBWINDOWS(tree, docnonce)` returns *subwindows*.

**AUXWINDOW.** This function takes a term *tree* and a document nonce *docnonce* as input as above. From all window terms in *tree* that have the window containing the document identified by *docnonce* as their opener, it selects one non-deterministically and returns its nonce. If there is no such window, it returns the nonce of the window containing *docnonce*.

**OPENERWINDOW.** This function takes a term *tree* and a document nonce *docnonce* as input as above. It returns the window nonce of the opener window of the window that contains

the document identified by *docnonce*. Recall that the nonce identifying the opener of each window is stored inside the window term. If no document with nonce *docnonce* is found in the tree *tree*,  $\diamond$  is returned.

**GETWINDOW.** This function takes a term *tree* and a document nonce *docnonce* as input as above. It returns the nonce of the window containing *docnonce*.

**GETORIGIN.** To extract the origin of a document, the function `GETORIGIN(tree, docnonce)` is used. This function searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It returns the origin *o* of the document. If no document with nonce *docnonce* is found in the tree *tree*,  $\diamond$  is returned.

### G.5.2 *script\_LPO\_cif*

As defined in Section 3.1, a script is a relation that takes as input a term and a set of nonces it may use. It outputs a new term. As specified in Section 3.4 (Triggering the Script of a Document ( $m = \text{TRIGGER}$ ,  $action = 1$ )) and formally specified in Algorithm 6, the input term is provided by the browser. It contains the current internal state of the script (which we call *scriptstate* in what follows) and additional information containing all browser state information the script has access to, such as the input the script has obtained so far via XHRs and postMessages, information about windows, etc. The browser expects the output term to have a specific form, as also specified in Section 3.4 and Algorithm 6. The output term contains, among other information, the new internal scriptstate.

As for *script\_LPO\_cif*, this script models the script run in the CIF, as sketched in Appendix F.

We first describe the structure of the internal scriptstate of the script *script\_LPO\_cif*.

**Definition 28.** A scriptstate *s* of *script\_LPO\_cif* is a term of the form  $\langle q, \text{parentOrigin}, \text{loggedInUser}, \text{pause}, \text{context}, \text{key}, \text{handledInputs}, \text{refXHRctx}, \text{refXHRcert} \rangle$  where  $q \in \mathbb{S}$ ,  $\text{parentOrigin} \in \text{Origins} \cup \{\perp\}$ ,  $\text{loggedInUser} \in \text{ID} \cup \{\langle \rangle, \perp\}$ ,  $\text{pause} \in \{\top, \perp\}$ ,  $\text{context} \in \mathcal{T}_{\mathcal{N}}$ ,  $\text{key} \in \mathcal{N} \cup \{\perp\}$ ,  $\text{handledInputs} \subset^{\diamond} \mathbb{N}$ ,  $\text{refXHRctx}, \text{refXHRcert} \in \mathcal{N} \cup \{\perp\}$ .

The initial state  $\text{initState}_{\text{cif}}$  of *script\_LPO\_cif* is the state  $\langle \text{init}, \perp, \perp, \perp, \perp, \perp, \langle \rangle, \perp, \perp \rangle$ .

Before we provide the formal specification of the relation that defines the behavior of *script\_LPO\_cif*, we present an informal description. The behavior mainly depends on the state *q* the script is in.

$q = \text{init}$  is the initial state. It's only transition handles no input and outputs a `postMessage cifready` to its parent window and transitions to `default`.

$q = \text{default}$  is the state to which *script\_LPO\_cif* always returns to. This state handles all `postMessages` the CIF expects to receive. If the `postMessage` received was sent from the parent window of the CIF, it behaves as follows:

**postMessage loaded** records the sender's origin of the received `postMessage` as the remote origin in the scriptstate. Also, an ID, which represents the assumption of the sender on who it believes to be logged in, is saved in the scriptstate. If the *pause* flag in the scriptstate is  $\top$  it transitions to the state `default`. Otherwise, it is checked, if the current context in the scriptstate is  $\perp$ . If the check is true, the script transitions to the state `fetchContext`, or to the state `checkAndEmit` otherwise.

**postMessage** `dlgRun` sets *pause* flag in the scriptstate to  $\top$  and transitions to **default**.

**postMessage** `dlgCmplt` sets the *pause* flag in the scriptstate to  $\perp$ . It then transitions to the state **fetchContext**.

**postMessage** `loggedInUser` has to contain an ID. This ID is saved in the scriptstate and then the script transitions to **default**.

**postMessage** `logout` removes the entry for the RP (recorded in the scriptstate) from the `localStorage` and then transitions to the state **sendLogout**.

$q = \text{fetchContext}$  sends an XHR to LPO with a GET request to the path `/ctx` and then transitions to the state **receiveContext**.

$q = \text{receiveContext}$  expects an XHR response as input containing the session context. This context is saved as the current context in the scriptstate. The script transitions to **checkAndEmit**.

$q = \text{checkAndEmit}$  lets the script transition to **requestUC** iff (1) some email address is marked as logged in at RP in the `localStorage`, (2) if an email address is recorded in the current scriptstate, this email address differs from the one recorded in the `localStorage`, and (3) the user is marked as logged in in the current context. Otherwise, if the email address recorded in the current scriptstate is  $\langle \rangle$ , the script transitions to **default**, else it transitions to **sendLogout**.

$q = \text{requestUC}$  creates a new private key (by taking a fresh nonce), stores the key in the scriptstate, and sends out an XHR POST request with the ID recorded in the `localStorage` for the parent window's origin and the public key (which can be derived from the private key) to LPO to get a UC. The script transitions to **receiveUC**.

$q = \text{receiveUC}$  receives an XHR response (from LPO) containing a UC. It creates an IA for the parent window's origin, combines the UC and the IA to a CAP, and sends the CAP as `login` postMessage to the parent window. The script then transitions back to the **default** state.

$q = \text{sendLogout}$  sends a `logout` postMessage to the parent document and then the script transitions to the **default** state.

We now specify the relation  $\text{script\_LPO\_cif} \subseteq (\mathcal{T}_{\mathcal{N}} \times 2^{\mathcal{N}}) \times \mathcal{T}_{\mathcal{N}}$  of the CIF's scripting process formally. Just like in Appendix B.2, we describe this relation by a non-deterministic algorithm.

Just like all scripts, as explained in Section 3.4 (see also Algorithm 6 for the formal specification), the input term this script obtains from the browser contains the cleaned tree of the browser's windows and documents *tree*, the nonce of the current document *docnonce*, its own scriptstate *scriptstate* (as defined in Definition 28), a sequence of all inputs *scriptinput* (also containing already handled inputs), a dictionary *cookies* of all accessible cookies of the document's domain, the `localStorage` *localStorage* belonging to the document's origin, the secrets *secret* of the document's origin, and a set *nonces* of fresh nonces as input. The script returns a new scriptstate  $s'$ , a new set of cookies  $\text{cookies}'$ , a new `localStorage`  $\text{localStorage}'$ , and a term *command* denoting a command to the browser.

---

**Algorithm 13** Relation of *script\_LPO\_cif*

---

**Input:**  $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, secret \rangle, nonces$

```

1: let  $s' := scriptstate$ 
2: let  $cookies' := cookies$ 
3: let  $localStorage' := localStorage$ 
4: switch  $s'.q$  do
5:   case init
6:     let  $command := \langle POSTMESSAGE, PARENTWINDOW(tree, docnonce), \langle cifready, \langle \rangle \rangle, \perp \rangle$ 
7:     let  $s'.q := default$ 
8:     stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
9:   case default
10:    let  $input, s' := CHOOSEINPUT(s', scriptinputs)$ 
11:    if  $\pi_1(input) \equiv POSTMESSAGE$  then
12:      let  $senderWindow := \pi_2(input)$ 
13:      let  $senderOrigin := \pi_3(input)$ 
14:      let  $m := \pi_4(input)$ 
15:      if  $senderWindow \equiv PARENTWINDOW(tree, docnonce)$  then
16:        switch  $m$  do
17:          case  $\langle loaded, id \rangle$ 
18:            let  $s'.parentOrigin := senderOrigin$ 
19:            let  $s'.loggedInUser := id$ 
20:            if  $s'.pause \equiv \top$  then
21:              stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
22:            else if  $s'.context \equiv \perp$  then
23:              let  $s'.q := fetchContext$ 
24:              stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
25:            else
26:              let  $s'.q := checkAndEmit$ 
27:              stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
28:            end if
29:          case  $\langle dlgRun, \langle \rangle \rangle$ 
30:            let  $s'.pause := \top$ 
31:            stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
32:          case  $\langle dlgCmplt, \langle \rangle \rangle$ 
33:            let  $s'.pause := \perp$ 
34:            let  $s'.q := fetchContext$ 
35:            stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
36:          case  $\langle loggedInUser, id \rangle$ 
37:            let  $s'.loggedInUser := id$ 
38:            stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
39:          case  $\langle logout, \langle \rangle \rangle$ 
40:        end if
41:      end if
42:    case fetchContext
43:      let  $s'.refXHRctx \leftarrow nonces$ 
44:      let  $command := \langle XMLHTTPREQUEST, URL_{ctx}^{LPO}, GET, \langle \rangle, s'.refXHRctx \rangle$ 
45:      let  $s'.q := receiveContext$ 
46:      stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
47:    case receiveContext
48:      let  $input, s' := CHOOSEINPUT(s', scriptinputs)$ 
49:      if  $(\pi_1(input) \equiv XMLHTTPREQUEST) \wedge (\pi_3(input) \equiv s'.refXHRctx)$  then
50:        let  $s'.context := \pi_2(input)$ 

```

```

51:     let s'.q := checkAndEmit
52:     stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
53:   end if
54:   case checkAndEmit
55:     let lid := localStorage'[siteInfo][s'.parentOrigin]
56:     if (lid ≠ ⟨⟩) ∧ (s'.loggedInUser ∉ {⟨⟩, ⊥}) ⇒ (s'.loggedInUser ≠ lid) ∧ (π1(s'.context) ≠ ⟨⟩)
then
57:     let s'.q := requestUC
58:     stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
59:   else if s'.loggedInUser ≡ ⟨⟩ then
60:     let s'.q := default
61:     stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
62:   else
63:     let s'.q := sendLogout
64:     stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
65:   end if
66:   case requestUC
67:     let id := localStorage'[siteInfo][s'.parentOrigin]
68:     let s'.key ← nonces
69:     let body := ⟨id, pub(s'.key), s'.context.xsrfToken⟩
70:     let s'.refXHRcert ← nonces \ {s'.key}
71:     let command := ⟨XMLHTTPREQUEST, URLLPO/certreq, POST, body, s'.refXHRcert⟩
72:     let s'.q := receiveUC
73:     stop ⟨s', cookies', localStorage', sessionStorage, command⟩
74:   case receiveUC
75:     let input, s' := CHOOSEINPUT(s', scriptinputs)
76:     if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRcert) then
77:       let uc := π2(input)
78:       let ia := sig(s'.parentOrigin, s'.key)
79:       let cap := ⟨uc, ia⟩
80:       let command :=
⟨POSTMESSAGE, PARENTWINDOW(tree, docnonce), ⟨login, cap⟩, s'.parentOrigin⟩
81:       let s'.q := default
82:       stop ⟨s', cookies', localStorage', sessionStorage, command⟩
83:     end if
84:   case sendLogout
85:     let command := ⟨POSTMESSAGE, PARENTWINDOW(tree, docnonce), ⟨logout, ⟨⟩⟩, ⊥⟩
86:     let s'.q := default
87:     stop ⟨s', cookies', localStorage', sessionStorage, command⟩
88: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩

```

### G.5.3 *script\_LPO\_ld*

The script *script\_LPO\_ld* models the script that runs in the LD. Its formal specification, presented next, follows the one presented above for *script\_LPO\_cif*.

**Definition 29.** A scriptstate  $s$  of *script\_LPO\_ld* is a term of the form  $\langle q, requestOrigin, context, key, handledInputs, refXHRctx, refXHRauth, refXHRcert \rangle$  with  $q \in \mathbb{S}$ ,  $requestOrigin \in \text{Origins} \cup \{\perp\}$ ,  $context \in \mathcal{T}_{\mathcal{N}}$ ,  $key \in \mathcal{N} \cup \{\perp\}$ ,  $handledInputs \subset^{\langle \rangle} \mathbb{N}$ ,  $refXHRctx, refXHRauth, refXHRcert \in \mathcal{N} \cup \{\perp\}$ .

The initial state  $initState_{ld}$  is the state  $\langle \mathbf{init}, \perp, \perp, \perp, \langle \rangle, \perp, \perp, \perp \rangle$ .

Before we provide the formal specification of the relation that defines the behavior of *script\_LPO\_ld*, we present an informal description. The behavior mainly depends on the state  $q$  the script is in.

$q \equiv \text{init}$  is the initial state. Its only transition takes no input and outputs a `postMessage` `ldready` to its parent window and transitions to `start`.

$q \equiv \text{start}$  expects a `request` `postMessage`. The sender's origin of this `postMessage` is recorded as the requesting origin in the `scriptstate`. An XHR is sent to LPO with a `GET` request to the path `/ctx` and then the script transitions to the state `receiveContext`.

$q \equiv \text{receiveContext}$  expects an XHR response as input containing the session context. This context is saved as the current context in the `scriptstate`. If the received context contains `\langle \rangle` as the ID list, the script transitions to the state `requestAuth`. Else, the script transitions to `requestUC`.

$q \equiv \text{requestAuth}$  sends an XHR `POST` request to LPO with the path `/auth` containing a browser's secret. The script then transitions to the state `receiveAuth`.

$q \equiv \text{receiveAuth}$  expects an XHR response as input containing  $\top$ . The script then sends an XHR to LPO with a `GET` request to the path `/ctx` and then transitions to the state `receiveContext`.

$q \equiv \text{requestUC}$  chooses (non-deterministically) an id, chooses a fresh private key and sends the id and the public key (corresponding to the private key) as an XHR `POST` request to LPO with the path `/certreq`. The script then transitions to `receiveUC`

$q \equiv \text{receiveUC}$  receive UC from LPO, create IA, combine with UC to CAP, record ID as logged in at the requester's origin. Send CAP in `postMessage` to parent. Go to state `null`

$q \equiv \text{null}$  do nothing.

We now formally specify the relation  $\text{script\_LPO\_ld} \subseteq (\mathcal{T}_{\mathcal{N}} \times 2^{\mathcal{N}}) \times \mathcal{T}_{\mathcal{N}}$  of the LD's scripting process. Just like in Appendix B.2, we describe this relation by a non-deterministic algorithm. Like all scripts, the input term given to this script is determined by the browser and the browser expects a term of a specific form (see Algorithm 6).

---

**Algorithm 14** Relation of *script\_LPO\_ld*


---

**Input:**  $\langle \text{tree}, \text{docnonce}, \text{scriptstate}, \text{scriptinputs}, \text{cookies}, \text{localStorage}, \text{sessionStorage}, \text{secret} \rangle, \text{nonces}$

```

1: let  $s' := \text{scriptstate}$ 
2: let  $\text{cookies}' := \text{cookies}$ 
3: let  $\text{localStorage}' := \text{localStorage}$ 
4: switch  $s'.q$  do
5:   case init
6:     let  $\text{command} := \langle \text{POSTMESSAGE}, \text{OPENERWINDOW}(\text{tree}, \text{docnonce}), \langle \text{ldready}, \langle \rangle \rangle, \perp \rangle$ 
7:     let  $s'.q := \text{start}$ 
8:     stop  $\langle s', \text{cookies}', \text{localStorage}', \text{sessionStorage}, \text{command} \rangle$ 
9:   case start
10:    let  $\text{input}, s' := \text{CHOOSEINPUT}(s', \text{scriptinputs})$ 
11:    if  $\pi_1(\text{input}) \equiv \text{POSTMESSAGE}$  then

```

```

12:     let senderWindow :=  $\pi_2(input)$ 
13:     let senderOrigin :=  $\pi_3(input)$ 
14:     let m :=  $\pi_4(input)$ 
15:     if  $m \equiv \langle request, \langle \rangle \rangle$  then
16:         let s'.requestOrigin := senderOrigin
17:         let s'.refXHRctx  $\leftarrow$  nonces
18:         let command :=  $\langle XMLHTTPREQUEST, URL_{/ctx}^{LPO}, GET, \langle \rangle, s'.refXHRctx \rangle$ 
19:         let s'.q := receiveContext
20:         stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
21:     end if
22: end if
23: case receiveContext
24:     let input, s' := CHOOSEINPUT(s', scriptinputs)
25:     if  $(\pi_1(input) \equiv XMLHTTPREQUEST) \wedge (\pi_3(input) \equiv s'.refXHRctx)$  then
26:         let s'.context :=  $\pi_2(input)$ 
27:         if  $\pi_1(s'.context) \equiv \langle \rangle$  then
28:             let s'.q := requestAuth
29:             stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
30:         else
31:             let s'.q := requestUC
32:             stop  $\langle s', cookies', localStorage', sessionStorage, \langle \rangle \rangle$ 
33:         end if
34:     end if
35: case requestAuth
36:     let body :=  $\langle secret, s'.context.xsrfToken \rangle$ 
37:     let s'.refXHRauth  $\leftarrow$  nonces
38:     let command :=  $\langle XMLHTTPREQUEST, URL_{/auth}^{LPO}, POST, body, s'.refXHRauth \rangle$ 
39:     let s'.q := receiveContext
40:     stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
41: case receiveAuth
42:     let input, s' := CHOOSEINPUT(s', scriptinputs)
43:     if  $(\pi_1(input) \equiv XMLHTTPREQUEST) \wedge (\pi_3(input) \equiv s'.refXHRauth)$  then
44:         if  $\pi_2(input) \equiv \top$  then
45:             let command :=  $\langle XMLHTTPREQUEST, URL_{/ctx}^{LPO}, GET, \langle \rangle \rangle$ 
46:             let s'.q := receiveContext
47:             stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
48:         end if
49:     end if
50: case requestUC
51:     let id  $\leftarrow$  s'.context.ids
52:     let s'.key  $\leftarrow$  nonces
53:     let body :=  $\langle id, pub(s'.key), s'.context.xsrfToken \rangle$ 
54:     let s'.refXHRcert  $\leftarrow$  nonces  $\setminus \{s'.key\}$ 
55:     let command :=  $\langle XMLHTTPREQUEST, URL_{/certreq}^{LPO}, POST, body, s'.refXHRcert \rangle$ 
56:     let s'.q := receiveUC
57:     stop  $\langle s', cookies', localStorage', sessionStorage, command \rangle$ 
58: case receiveUC
59:     let input, s' := CHOOSEINPUT(s', scriptinputs)
60:     if  $(\pi_1(input) \equiv XMLHTTPREQUEST) \wedge (\pi_3(input) \equiv s'.refXHRcert)$  then
61:         let uc :=  $\pi_2(input)$ 
62:         let ia := sig(s'.requestOrigin, s'.key)
63:         let cap :=  $\langle uc, ia \rangle$ 

```

```

64:      let command :=
          ⟨POSTMESSAGE, OPENERWINDOW(tree, docnonce), ⟨response, cap⟩, s'.requestOrigin⟩
65:      let s'.q := null
66:      stop ⟨s', cookies', localStorage', sessionStorage, command⟩
67:    end if
68: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩

```

#### G.5.4 *script\_RP\_index*

The script *script\_RP\_index* models the script that is run by an RP. Its formal specification, presented next, follows the one presented for the other scripts above.

**Definition 30.** A *scriptstate*  $s$  of *script\_RP\_index* is a term of the form  $\langle q, CIFindex, LDindex, dialogRunning, cap, handledInputs, refXHRcap \rangle$  with  $q \in \mathbb{S}$ ,  $CIFindex \in \mathbb{N} \cup \{\perp\}$ ,  $dialogRunning \in \{\top, \perp\}$ ,  $cap \in \mathcal{T}_{\mathcal{N}}$ ,  $handledInputs \subset^{\langle \rangle} \mathbb{N}$ ,  $refXHRcap \in \mathcal{N} \cup \{\perp\}$ .

We call  $s$  the *initial scriptstate* of *script\_RP\_index* iff  $s \equiv \langle \mathbf{init}, \perp, \perp, \perp, \langle \rangle, \langle \rangle, \perp \rangle$ .

Before we provide the formal specification of the relation that defines the behavior of *script\_RP\_index*, we present an informal description. The behavior mainly depends on the state  $q$  the script is in.

$q \equiv \mathbf{init}$  is the initial state. It creates the CIF iframe and then transitions to **receiveCIFReady**.

$q \equiv \mathbf{receiveCIFReady}$  expects a **cifready** postMessage from the CIF iframe with origin of LPO. It chooses some ID,  $\langle \rangle$ , or  $\perp$  and sends this as a **loaded** postMessage to the CIF iframe with receiver's origin set to the origin of LPO.<sup>24</sup> It then transitions to the state **default**.

$q \equiv \mathbf{default}$  chooses non-deterministically between (1) opening the LD subwindow and then transitions to the same state or (2) handling one of the following postMessages:

**postMessage login** which has to be sent from the CIF with origin of LPO. Handling this postMessage stores the CAP (contained in the postMessage) in the scriptstate and then transitions to the **sendCAP** state.

**postMessage logout** which has to be sent from the CIF with origin of LPO. Handling this postMessage has no effect and results in the same state.

**postMessage ldready** which can only be handled after the LD has been opened and before a **response** postMessage has been received. The **ldready** postMessage has to be sent from the origin of LPO. The script sends a **request** postMessage to the LD and stays in the **default** state.

**postMessage response** which can only be handled after the LD has been opened and before another **response** postMessage has been received. The **ldready** postMessage has to be sent from the origin of LPO. Handling this postMessage stores the CAP (contained in the postMessage) in the scriptstate, closes the LD, and then transitions to the **dlgClosed** state.

---

<sup>24</sup>From the point of view of the real scripts running at RP either some ID is considered to be logged in (e.g. from some former "session"), or that no one is considered to be logged in ( $\langle \rangle$ ), or that *script\_RP\_index* does not know if it should consider someone to be logged in ( $\perp$ ). This is overapproximated here by allowing *script\_RP\_index* to choose non-deterministically between these cases.

$q \equiv \text{dlgClosed}$  sends a `loggedInUser` `postMessage` to the CIF and transitions to the state `loggedInUser`.

$q \equiv \text{loggedInUser}$  sends a `dlgCmplt` `postMessage` to the CIF and transitions to the state `sendCAP`.

$q \equiv \text{sendCAP}$  sends the CAP to RP as a `POST XHR` and then transitions to the state `receiveServiceToken`.

$q \equiv \text{receiveServiceToken}$  receives  $\langle n, i \rangle$  from RP, but does not do anything with it. The script then transitions to the state `default`.

We now formally specify the relation  $\text{script\_RP\_index} \subseteq (\mathcal{T}_{\mathcal{N}} \times 2^{\mathcal{N}}) \times \mathcal{T}_{\mathcal{N}}$  of the RP-Doc's scripting process. Just like in Appendix B.2, we describe this relation by a non-deterministic algorithm. Like all scripts, the input term given to this script is determined by the browser and the browser expects a term of a specific form (see Algorithm 6). Following Algorithm 15, we provide some more explanation.

---

**Algorithm 15** Relation of  $\text{script\_RP\_index}$ 


---

**Input:**  $\langle \text{tree}, \text{docnonce}, \text{scriptstate}, \text{scriptinputs}, \text{cookies}, \text{localStorage}, \text{sessionStorage}, \text{secret} \rangle, \text{nonces}$

```

1: let  $s' := \text{scriptstate}$ 
2: let  $\text{cookies}' := \text{cookies}$ 
3: let  $\text{localStorage}' := \text{localStorage}$ 
4: switch  $s'.q$  do
5:   case init
6:     let  $\text{command} := \langle \text{IFRAME}, \text{URL}_{/\text{cif}}^{\text{LPO}}, \text{GETWINDOW}(\text{tree}, \text{docnonce}) \rangle$ 
7:     let  $s'.q := \text{receiveCIFReady}$ 
8:     let  $\text{subwindows} := \text{SUBWINDOWS}(\text{tree}, \text{docnonce})$ 
9:     let  $s'.\text{CIFindex} := |\text{subwindows}| + 1$   $\triangleright$  Index of the next subwindow to be created.
10:    stop  $\langle s', \text{cookies}', \text{localStorage}', \text{sessionStorage}, \text{command} \rangle$ 
11:   case receiveCIFReady
12:     let  $\text{input}, s' := \text{CHOOSEINPUT}(s', \text{scriptinputs})$ 
13:     if  $\pi_1(\text{input}) \equiv \text{POSTMESSAGE}$  then
14:       let  $\text{senderWindow} := \pi_2(\text{input})$ 
15:       let  $\text{senderOrigin} := \pi_3(\text{input})$ 
16:       let  $m := \pi_4(\text{input})$ 
17:       let  $\text{subwindows} := \text{SUBWINDOWS}(\text{tree}, \text{docnonce})$ 
18:       if  $(m \equiv \langle \text{cifready}, \langle \rangle \rangle) \wedge (\text{senderOrigin} \equiv \text{origin}_{\text{LPO}}) \wedge$   

 $(\text{senderWindow} \equiv \pi_{s'.\text{CIFindex}}(\text{subwindows}).\text{nonce})$  then
19:         let  $\text{id} \leftarrow \{ \perp, \langle \rangle \} \cup \text{ID}$ 
20:         let  $\text{command} := \langle \text{POSTMESSAGE}, \pi_{s'.\text{CIFindex}}(\text{subwindows}), \langle \text{loaded}, \text{id} \rangle, \text{origin}_{\text{LPO}} \rangle$ 
21:         let  $s'.q := \text{default}$ 
22:         stop  $\langle s', \text{cookies}', \text{localStorage}', \text{sessionStorage}, \text{command} \rangle$ 
23:       end if
24:     end if
25:   case default
26:     if  $s'.\text{dialogRunning} \equiv \perp$  then
27:       let  $\text{choice} \leftarrow \{ \text{openLD}, \text{handlePM} \}$ 
28:     else
29:       let  $\text{choice} := \text{handlePM}$ 
30:     end if

```

```

31:   if choice  $\equiv$  openLD then
32:     let s'.dialogRunning :=  $\top$ 
33:     let command :=  $\langle$ HREF, URLldLPO, _BLANK $\rangle$ 
34:     let s'.q := default
35:     stop  $\langle$ s', cookies', localStorage', sessionStorage, command $\rangle$ 
36:   else
37:     let input, s' := CHOOSEINPUT(s', scriptinputs)
38:     if  $\pi_1$ (input)  $\equiv$  POSTMESSAGE then
39:       let senderWindow :=  $\pi_2$ (input)
40:       let senderOrigin :=  $\pi_3$ (input)
41:       let m :=  $\pi_4$ (input)
42:       let subwindows := SUBWINDOWS(tree, docnonce)
43:       if senderOrigin  $\equiv$  originLPO then
44:         if senderWindow  $\equiv$   $\pi_{s'}$ .CIFindex(subwindows).nonce then
45:           if  $\pi_1$ (m)  $\equiv$  login then
46:             let s'.cap :=  $\pi_2$ (m)
47:             let s'.q := sendCAP
48:             stop  $\langle$ s', cookies', localStorage', sessionStorage,  $\langle$  $\rangle$  $\rangle$ 
49:           else if  $\pi_1$ (m)  $\equiv$  logout then
50:             let s'.q := default
51:             stop  $\langle$ s', cookies', localStorage', sessionStorage,  $\langle$  $\rangle$  $\rangle$ 
52:           end if
53:         else if s'.dialogRunning  $\equiv$   $\top$  then
54:           if  $\pi_1$ (m)  $\equiv$  ldready then
55:             let command :=
56:                $\langle$ POSTMESSAGE, AUXWINDOW(tree, docnonce),  $\langle$ request,  $\langle$  $\rangle$  $\rangle$ , originLPO $\rangle$ 
57:             let s'.q := default
58:             stop  $\langle$ s', cookies', localStorage', sessionStorage, command $\rangle$ 
59:           else if  $\pi_1$ (m)  $\equiv$  response then
60:             let s'.dialogRunning :=  $\perp$ 
61:             let s'.cap :=  $\pi_2$ (m)
62:             let command :=  $\langle$ CLOSE, AUXWINDOW(tree, docnonce) $\rangle$ 
63:             let s'.q := dlgClosed
64:             stop  $\langle$ s', cookies', localStorage', sessionStorage, command $\rangle$ 
65:           end if
66:         end if
67:       end if
68:     end if
69:   case dlgClosed
70:     let subwindows := SUBWINDOWS(tree, docnonce)
71:     let id :=  $\pi_1$ (extractmsg( $\pi_1$ (s'.cap)))  $\triangleright$  Extract ID from CAP.
72:     let command :=  $\langle$ POSTMESSAGE,  $\pi_{s'}$ .CIFindex(subwindows).nonce,  $\langle$ loggedInUser, id $\rangle$ , originLPO $\rangle$ 
73:     let s'.q := loggedInUser
74:     stop  $\langle$ s', cookies', localStorage', sessionStorage, command $\rangle$ 
75:   case loggedInUser
76:     let subwindows := SUBWINDOWS(tree, docnonce)
77:     let command :=  $\langle$ POSTMESSAGE,  $\pi_{s'}$ .CIFindex(subwindows).nonce,  $\langle$ dlgCmplt,  $\langle$  $\rangle$  $\rangle$ , originLPO $\rangle$ 
78:     let s'.q := sendCAP
79:     stop  $\langle$ s', cookies', localStorage', sessionStorage, command $\rangle$ 
80:   case sendCAP
81:     let s'.refXHRcap  $\leftarrow$  nonces

```

```

82:   let host, protocol such that ⟨host, protocol⟩ = GETORIGIN(tree, docnonce) if possible; otherwise stop ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
83:   let command := ⟨XMLHTTPREQUEST, ⟨URL, protocol, host, /, ⟨⟩⟩, POST, s'.cap, s'.refXHRcap⟩ ▷
Relay received CAP to RP.
84:   let s'.q := receiveServiceToken
85:   stop ⟨s', cookies', localStorage', sessionStorage, command⟩
86: case receiveServiceToken
87:   let input, s' := CHOOSEINPUT(s', scriptinputs)
88:   if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRcap) then
89:     let s'.q := default
90:     stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
91:   end if
92: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩

```

In Lines 6–10 and 32–35 the script asks the browser to create iframes. To obtain the window reference for these iframes, the script first determines the current number of subwindows and stores it (incremented by 1) in the scriptstate (`CIFindex` and `LDindex`, respectively). When the script is invoked the next time, the iframe the script asked to be created will have been added to the sequence of subwindows by the browser directly following the previously existing subwindows. The script can therefore access the iframe by the indexes `CIFindex` and `LDindex`, respectively.

## H Security Property

Formally, the security property for BrowserID is defined as follows. First note that every RP service token  $\langle n, i \rangle$  recorded in RP was created by RP as the result of a unique HTTPS POST request  $m$  with a valid CAP for ID  $i$ . We refer to  $m$  as the *request corresponding to*  $\langle n, i \rangle$ .

**Definition 31.** Let  $\mathcal{BID}$  be a BrowserID web system. We say that  $\mathcal{BID}$  is secure if for every run  $\rho$  of  $\mathcal{BID}$ , every state  $(S_j, E_j)$  in  $\rho$ , every  $r \in \text{RP}$ , every RP service token of the form  $\langle n, i \rangle$  recorded in  $r$  in the state  $S_j(r)$ , the following two conditions are satisfied:

(A) If  $\langle n, i \rangle$  is derivable from the attackers knowledge in  $S_j$  (i.e.,  $\langle n, i \rangle \in d_{N^{\text{attacker}}}(S_j(\text{attacker}))$ ), then it follows that the browser owning  $i$  is fully corrupted in  $S_j$ , i.e., the value of `isCorrupted` is `FULLCORRUPT`.

(B) If the request corresponding to  $\langle n, i \rangle$  was sent by some  $b \in \mathbf{B}$  which is honest in  $S_j$ , then  $b$  owns  $i$ .

## I Proof of Theorem 1

In order to prove Theorem 1, we have to prove Conditions A and B of Definition 31. These are proven separately in what follows:

### I.1 Condition A

We assume that Condition A is not satisfied and prove that this leads to a contradiction. That is, we make the following assumption (\*): There is a run  $\rho = s_0, s_1, \dots$  of  $\mathcal{BID}$ , a state  $s_j = (S_j, E_j)$  in  $\rho$ , an  $r \in \text{RP}$ , an RP service token of the form  $\langle n, i \rangle$  recorded in  $r$  in the state  $S_j(r)$  such that  $\langle n, i \rangle \in d_{N^{\text{attacker}}}(S_j(\text{attacker}))$  and the browser owning  $i$  is not fully corrupted in  $S_j$ .

Without loss of generality, we may assume that  $\rho$  also satisfies the following:

(\*\*) Whenever a browser becomes corrupted (i.e., either FULLCORRUPT or CLOSECORRUPT) in a processing step leading to some state  $s_l$  in  $\rho$ , this browser is triggered immediately afterwards again (in the processing step leading to  $s_{l+1}$ ) and sends the full state of the web browser to the attacker process **attacker**, which then receives this knowledge in state  $s_{l+2}$ . Afterwards, this browser is not triggered anymore.

(\*\*\*) For every term  $\text{enc}_a(t, \text{pub}(k'))$  for some  $t \in \mathcal{T}_{\mathcal{N}}$ ,  $k' \in \mathcal{N}$  that is a subterm of the output of a transition of  $R^{\text{att}}$  but not of the input, i.e.,  $R^{\text{att}}$  has created  $\text{enc}_a(t, k')$  by itself,  $R^{\text{att}}$  has sent an HTTP message containing  $t$  (unencrypted) to some  $d \in \text{dom}(\text{attacker})$  before.

If there is a run that satisfies (\*), it is easy to turn this run into a run that satisfies both (\*) and (\*\*). This is because an attacker who obtains the state of the browser can simulate the browser himself. Moreover, it is easy to turn a run that satisfies (\*) and (\*\*) but not (\*\*\*) into a run that satisfies all three properties by adding the necessary requests from the script  $R^{\text{att}}$ .

Given (\*), by definition of RPs, for  $\langle n, i \rangle$  there exists a corresponding HTTPS request received by  $r$ , which we call  $\text{req}_{\text{cap}}$ , and a corresponding response  $\text{resp}_{\text{cap}}$ . The request must contain a valid CAP  $c$  and must have been sent by some atomic process  $p$  to  $r$ . The response must contain  $\langle n, i \rangle$  and it must be encrypted by some symmetric encryption key  $k$  sent in  $\text{req}_{\text{cap}}$ .

In particular, it follows that the request and the response must be of the following form, where  $d_r = \text{dom}(r)$  is the domain of  $r$ ,  $n_{\text{cap}}, k \in \mathcal{N}$  are some nonces, and  $c$  is some valid CAP:

$$\text{req}_{\text{cap}} = \text{enc}_a(\langle \langle \text{HTTPReq}, n_{\text{cap}}, \text{POST}, d_r, /, \langle \rangle, [\text{Origin} : \langle d_r, \mathbf{S} \rangle], c \rangle, k \rangle, \text{pub}(\text{key}(d_r))) , \quad (17)$$

$$\text{resp}_{\text{cap}} = \text{enc}_s(\langle \langle \text{HTTPResp}, n_{\text{cap}}, 200, \langle \rangle, \langle n, i \rangle \rangle, k) . \quad (18)$$

Moreover, there must exist a processing step of the following form where  $m \leq j$ ,  $a_r \in \text{addr}(r)$ , and  $x$  is some address:

$$s_{m-1} \xrightarrow[r \rightarrow \{(x:a_r:\text{resp}_{\text{cap}})\}]{(a_r:x:\text{req}_{\text{cap}}) \rightarrow r} s_m .$$

From the assumption and the definition of RPs it follows that  $c$  is issued for  $d_r$  (otherwise, RP would not accept the CAP, see Line 13 of Algorithm 11). The nonce  $n$  in  $\langle n, i \rangle$  is chosen freshly and from RPs nonces  $N^r$ . It is not used again by  $r$  afterwards.

We assume that  $s_j$  is the *first* state in  $\rho$  where  $\langle n, i \rangle \in d_{N^{\text{attacker}}}(S_j(\text{attacker}))$  (i.e., there is no  $j' < j$ ,  $\langle n, i \rangle \in d_{N^{\text{attacker}}}(S_{j'}(\text{attacker}))$ ).

We note that, by definition of attacker processes, the attacker never discards any information, i.e.,  $t \in d_{N^{\text{attacker}}}(S_u(\text{attacker}))$  implies  $t \in d_{N^{\text{attacker}}}(S_{u+1}(\text{attacker}))$  for every term  $t$  and  $u \in \mathbb{N}$ .

To conclude the proof, we now first prove several lemmas.

In what follows, given an atomic process  $p$  and a message  $m$ , we say that  $p$  *emits*  $m$  in a run  $\rho = s_0, s_1, \dots$  if there is a processing step of the form

$$s_{u-1} \xrightarrow[p \rightarrow E]{} s_u$$

for some  $u \in \mathbb{N}$ , set of events  $E$  and some addresses  $x, y$  with  $(x:y:m) \in E$ .

We say that an atomic process  $p$  *created* a message  $m$  (at some point) in a run if  $m$  is (congruent to) a subterm of a message emitted by  $p$  in some processing step and if there is no earlier processing step where  $m$  is a subterm of a message emitted by an atomic process  $p'$ .

We say that a browser  $b$  *accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function **PROCESSRESPONSE**, passing the message and the request (see Algorithm 7).

We say that an atomic DY process  $p$  *knows a term  $t$*  in some state  $s = (S, E)$  of a run if it can derive the term from its knowledge, i.e.,  $t \in d_{NP}(S(p))$ .

We say that a *script initiated a request  $r$*  if a browser triggered the script (in Line 12 of Algorithm 6) and the first component of the *command* output of the script relation is either HREF, IFRAME, FORM, or XMLHTTPREQUEST such that the browser issues the request  $r$  in the same step as a result.

**Lemma 1.** *If in a run  $\rho$  of  $\mathcal{BID}$  an honest browser  $b$  emits an HTTPS request of the form*

$$\text{enc}_a(\langle \text{req}, k \rangle, \text{pub}(k'))$$

where  $\text{req}$  is an HTTP request,  $k$  is a nonce (symmetric key),  $k'$  is the private key of an RP or of LPO, and if in that run  $b$  does not become fully corrupted, then all of the following statements are true:

1. *There is no state of  $\mathcal{BID}$  where any party except for  $b$  and the owner of the key  $k'$  knows  $k'$ , thus no one except for  $b$  and the owner of the key  $k'$  can decrypt  $\text{req}$ .*
2. *There is no state in the run  $\rho$  where  $k$  is known to any atomic process  $p$  (i.e., in no state  $s = (S, E)$  in  $\rho$ ,  $k \in d_{NP}(S(p))$ ), except for the atomic processes  $b$  and the owner of the public key (some RP or LPO).*
3. *The value of the host header in  $\text{req}$  is a domain of the owner of the key  $k'$ .*
4. *Only the owner of the key  $k'$  can create a response  $r$  to this request that is accepted by  $b$ , i.e. the nonce of the HTTP request is not known to any atomic process  $p$ , except for the atomic process  $b$  and the owner of the public key (some RP or LPO).*

*Proof.* First, we note that only the intended receiver can decrypt the message: The private keys of RPs and LPO are per definition only known to the respective parties. According to the definition of RPs and LPO, the keys do not leak to other parties, i.e., there is no state in a run  $\rho$  of  $\mathcal{BID}$  where the keys are known to any other parties except their respective owners. This proves (1).

We can further see from the definition of the receivers (some RP or LPO) that they use the key  $k$  only to encrypt the responses (Algorithms 10 and 11). In both definitions,  $k$  is extracted from the message and discarded after encrypting the response. Note that neither RPs nor LPO can be corrupted. Hence, neither RPs nor LPO can leak  $k$ . From the definition of the browser  $b$ , we see that the key is always chosen from a fresh set of nonces (Line 61 of Algorithm B.2.2) that are not used anywhere else. Further, the key is stored in the browser's state in *pendingRequests*, but discarded after receiving the response. The information from *pendingRequests* is not extracted or used anywhere else (in particular it is not accessible by scripts). If the browser becomes closecorrupted at some point in the run  $\rho$ , the key cannot be used anymore (compare Line 9 of Algorithm 8). Hence,  $k$  cannot leak from  $b$  either. This proves (2).

From Line 63 of Algorithm 8 we can see that the encryption key for the request  $\text{req}$  was actually chosen using the host header of the message. The mapping from domains to encryption keys in  $\mathcal{BID}$  is always "correct", i.e., the owner of  $k'$  is the owner of the domain that is given in the host header. This proves (3).

An HTTPS response  $r$  that is accepted by  $b$  as a response to the above request has to be encrypted with  $k$ . (This is checked by the browser using the *pendingRequests* state information

that is not alterable by scripts or other browser actions.) This nonce, however, is only known to the owner of the public key and  $b$ . The browser  $b$  cannot send responses. This proves (4).  $\square$

**Lemma 2.** *If in a run  $\rho$  of  $\mathcal{BID}$  an honest browser  $b$  has a document  $d$  in its state with the origin  $\langle dom, \mathcal{S} \rangle$  where  $dom \in \text{dom}(\text{RP}) \cup \text{dom}(\text{LPO})$ , then  $b$  extracted (in Line 33 in Algorithm 7) the script in that document from an HTTPS response that was emitted by the owner of the private key belonging to  $dom$ .*

*Proof.* The origin of the document  $d$  is set only once: In Line 33 of Algorithm 7. The values (domain and protocol) used there stem from the information about the request (say,  $req$ ) that led to loading of  $d$ . These values have been stored in *pendingRequests* between the request and the response actions. The contents of *pendingRequests* are indexed by freshly chosen nonces and can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request  $req$  was added to *pendingRequests* in Line 62 (or Line 65 which we can exclude as we will see later) of Algorithm 8. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in *pendingRequests*. When receiving the response to  $req$ , it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say,  $n$ ). Only then the protocol part of the origin of the newly created document becomes  $\mathcal{S}$ . The domain part of the origin (in our case  $dom$ ) is taken directly from the *pendingRequests* and is thus guaranteed to be unaltered.

From Line 63 of Algorithm 8 we can see that the encryption key for the request  $req$  was actually chosen using the host header of the message which will finally be the value of the origin of the document  $d$ . Since the honest browsers in  $\mathcal{BID}$  select the correct public keys for a domain, we can see that  $req$  was encrypted using the public key belonging to  $dom$ . With Lemma 1 we see that the symmetric encryption key for the response,  $k$ , is only known to  $b$  and the respective RP or LPO. The same holds for the nonce  $n$  that was chosen by the browser and included in the request. Thus, no other party than the owner of the private key belonging to  $dom$  can encrypt a response that is accepted by the browser  $b$  and which finally defines the script of the newly created document.  $\square$

**Lemma 3.** *If in a run  $\rho$  of  $\mathcal{BID}$  an honest browser  $b$  issues an HTTP(S) request with the Origin header value  $\langle dom, \mathcal{S} \rangle$  where  $dom \in \text{dom}(\text{RP}) \cup \text{dom}(\text{LPO})$ , then that request was initiated by a script that  $b$  extracted (in Line 33 in Algorithm 7) from an HTTPS response that was emitted by the owner of the private key belonging to  $dom$ .*

*Proof.* First, we can see that the request was initiated by a script: As it contains an origin header, it must have been a POST request (see the browser definition in Appendix B.2). POST requests can only be initiated in Lines 51, 74 of Algorithm 6 and Line 28 of Algorithm 7. In the latter instance (Location header redirect), the request contains at least two different origins, therefore it is impossible to create a request with exactly the origin  $\langle dom, \mathcal{S} \rangle$  using a redirect. In the other two cases (FORM and XMLHttpRequest), the request was initiated by a script.

The Origin header of the request is defined by the origin of the script's document. With Lemma 2 we see that the content of the document, in particular the script, was indeed provided by the owner of the private key belonging to  $dom$ .  $\square$

**Lemma 4.** *In a run  $\rho$  of  $\mathcal{BID}$ , if LPO sends a `browserid_state` cookie in a Set-Cookie header in an HTTPS response to an HTTPS request emitted by a browser  $b$ , there is no state in the run where the browser is honest and the attacker can derive the cookie value from its own knowledge.*

*Proof.* We can see that the browser is honest when sending the request (otherwise, it would not do so, (\*\*)). With Lemma 1 and as in the proof for Lemma 2 we see that the sender of the request to LPO (say,  $req$ ) is the same as the receiver, namely browser  $b$ . As the message is transferred over HTTPS, the attacker cannot read the cookie from the response.

The `browserid_state` cookie is sent to  $b$  as an `httpOnly` secure session cookie (compare Line 38 in Algorithm 10). When the response arrives at  $b$ , the cookie is transferred to the cookie store (Line 5 of Algorithm 7) which is indexed by domains. The cookie information can be accessed by scripts (Line 4 of Algorithm 6) and can be added to requests (Line 5 of Algorithm 5). As the `browserid_state` cookie is an `httpOnly` cookie we can rule out the first case. In the second case, the cookie can only be added to requests to the origin  $\langle \text{dom}(\text{LPO}), \mathbf{S} \rangle$ , as the cookie is marked as secure (as defined in Line 5 in Algorithm 5). These properties hold as long as the browser is not corrupted.

As a last step, we have to rule out that LPO or the browser use a cookie value that is known to the attacker via some other way. We will see that any cookie value was initially chosen by LPO.

First, we can see that the cookie value was either in the browser's knowledge before it received the `browserid_state` header or that it was chosen freshly by LPO. The only line where LPO sets the cookie is in Line 38 of Algorithm 10. From the lines before, it is easy to see that the session value that finally becomes the cookie value was either provided as a cookie in the request or is chosen from the set of unused nonces. In Lemma 6 we see that any value that is contained in a request sent by an honest browser to LPO was initially chosen by LPO.

We see that the attacker cannot know the cookie value as long as the browser stays honest, which proves the lemma.  $\square$

**Lemma 5.** *In every state  $s = (S, E)$  of run  $\rho$  of  $\mathcal{BID}$ , for every  $xsrftoken$  of an LPO session and its session ID  $sessionid$ , if  $xsrftoken \in d_{N_{\text{attacker}}}(S(\text{attacker}))$ , then  $sessionid \in d_{N_{\text{attacker}}}(S(\text{attacker}))$ , i.e., an attacker can only know an  $xsrftoken$  value for an LPO session if he knows the `browserid_state` session ID of that session.*

*Proof.* The  $xsrftoken$  is chosen by LPO (Line 34 in Algorithm 10). If LPO receives a POST request with the path `/ctx` that contains a `browserid_state` cookie containing a  $sessionid$  that is in its list of valid sessions, it returns  $xsrftoken$  as part of the response. If LPO receives a request to the same URL without a valid session ID, it creates a new session and returns  $sessionid$  as well as a freshly chosen  $xsrftoken$  in the response. For other requests (to other URLs, etc.)  $xsrftoken$  is not a part of the response at all.

The  $xsrftoken$  is only transferred over HTTPS: LPO only reacts to HTTPS requests (Line 20 of Algorithm 10) and the request that is sent from the browser to LPO to retrieve  $xsrftoken$  is explicitly sent over HTTPS (Line 44 of Algorithm 13 or Line 18 of Algorithm 14). Thus, if an honest browser sends a request to LPO, the attacker cannot read the response if the browser stays honest (Lemma 1). If the browser becomes corrupted later, the attacker learns the  $sessionid$  and the  $xsrftoken$  at the same time. The LPO script that has access to  $xsrftoken$  in the browser's state does not send out this part of the state to origins other than LPO's (see Algorithm 13 and Algorithm 14) and the  $xsrftoken$  is stored only temporarily in the script's state (as part of the *context*, see Line 50 in Algorithm 13 and Line 26 in Algorithm 14), such that it is never released when the browser is honest or closecorrupted.

We can see that the attacker knows  $sessionid$  whenever he knows  $xsrftoken$ , which proves the lemma.  $\square$

**Lemma 6.** *In a run  $\rho$  of  $\mathcal{BID}$ , for any HTTPS request  $req$  that is emitted by an honest browser  $b$  and that is encrypted with the public key of LPO, if there is a `Cookie` header in  $req$  containing a cookie with the name `browserid_state`, then there is an HTTPS response that was emitted by LPO previously in the run and that was accepted by  $b$ . In this response, a `Set-Cookie` header was sent with the name `browserid_state` and the same value as the `browserid_state` cookie in  $req$ .*

*Proof.* The cookie that is sent in  $req$  was taken from the cookie list that is stored in the browser state (see Algorithm 5). Cookies are stored per-domain, i.e.,  $\text{dom}(\text{LPO})$  in this case. Adding a cookie to this list can be achieved by adding a `Set-Cookie` to a response on a request to  $\text{dom}(\text{LPO})$  or by setting the cookie from a script in a document with the origin  $\langle \text{dom}(\text{LPO}), x \rangle$  where  $x \in \{\mathcal{P}, \mathcal{S}\}$ . The domain  $\text{dom}(\text{LPO})$  is part of the *sts* list in honest browsers (see Section G.2) thus the browser  $b$  never contacts the insecure origin  $\langle \text{dom}(\text{LPO}), \mathcal{P} \rangle$ . Thus, responses and scripts can only be received from the origin  $\langle \text{dom}(\text{LPO}), \mathcal{S} \rangle$  (see Lemma 1 Property (4) and Lemma 2). The LPO scripts `script_LPO_cif` and `script_LPO_ld` do not set cookies, thus the only possible way that a cookie can be stored in the browser's list of cookies is when LPO adds a `Set-Cookie` header to a HTTPS response. Obviously, this header has to have the same value as the cookie that is finally returned to the server. This proves the lemma.  $\square$

**Lemma 7.** *In a run  $\rho$  of  $\mathcal{BID}$ , if an honest browser  $b$  emits a request  $req_{\text{auth}}$  that is received by LPO and leads to the authentication<sup>25</sup> of an LPO session identified by the sessionid  $sessionid$ , then the identity  $i$ , for which the session was authenticated, is owned by  $b$ , i.e.,  $i \in \text{ownerOfID}^{-1}(b)$ .*

*Proof.* For authentication, a request of the following form has to be received by LPO:

$$\begin{aligned} req_{\text{auth}} = \text{enc}_a(\langle \langle \text{HTTPReq}, n_2, \text{POST}, \text{dom}(\text{LPO}), /auth, \langle \rangle, \\ \langle [\text{Cookie} : [\text{browserid\_state} : sessionid], \dots \rangle], \\ \langle s, xsrfToken \rangle, \rangle, k'' \rangle, \text{pub}(\text{key}(\text{dom}(\text{LPO})))) \end{aligned} \quad (19)$$

The request  $req_{\text{auth}}$  contains the secret  $s$  and the `xsrfToken` that, by definition of LPO, is stored at LPO along with `sessionid`. In an honest browser (which  $b$  is), this request can only be caused by a script (or through a redirection, which again would require a script to initiate the request in the first place). There are three scripts that can issue such a request: the attacker script and both LPO scripts. In the latter case, the LPO scripts will provide the browser secret as  $s$  and hence, authenticate for an identity owned by the browser. In the former case, the attacker script needs to know `xsrfToken`. Hence, by Lemma 5 he needs to know `sessionid`. However, the `sessionid` value does not leak from the honest browser  $b$  (Lemma 4) and cannot be set by the attacker (Lemma 6). Hence, the attacker cannot know `sessionid`, and hence, by Lemma 5 he cannot know `xsrfToken`, and hence,  $req_{\text{auth}}$  cannot have been initiated by the attacker script.  $\square$

**Lemma 8.** *In a run  $\rho$  of  $\mathcal{BID}$  if either `script_LPO_cif` or `script_LPO_ld` were loaded into a document with HTTPS origin and are used to create a CAP  $c$ , i.e., if  $c$  is contained in a `postMessage` that is sent in Line 82 of Algorithm 13 or in Line 66 of Algorithm 14, then the origin for which  $c$  is issued is the origin of the script that receives this `postMessage`.*

<sup>25</sup>See Section G.3 for an explanation on the authentication at LPO.

*Proof.* Looking at the case when *script\_LPO\_cif* issues the CAP in Line 80 of Algorithm 13, the origin for which the IA is issued in this case is determined by the element *s'.parentOrigin* of the script's state. This element is only written to in Line 18 of Algorithm 13. Its value is the sender origin of the *postMessage* requesting the CAP. The very same value determines the only allowed receiver origin of the *postMessage* that returns the CAP (Line 80). With a very similar argument (different line numbers), we can see that the statement for *script\_LPO\_ld* holds true as well.  $\square$

**Lemma 9.** *In a run  $\rho$  of  $\mathcal{BID}$ , if a CAP  $c = \langle uc, ia \rangle$  is sent by *script\_RP\_index* (Line 85 of Algorithm 15) running in an honest browser  $b \in \mathbf{B}$  in a document with origin  $\langle d_r, \mathbf{S} \rangle$  as an HTTP(S) message to an RP  $r \in \mathbf{RP}$ , where  $d_r = \text{dom}(r)$ ,  $uc = \text{sig}(\langle i, \text{pub}(k_u) \rangle, k^{\text{LPO}})$ , and  $ia = \text{sig}(o, k'_u)$ , then all of the following statements are true:*

1.  *$c$  is a valid CAP. In particular,  $k_u = k'_u$ .*
2.  *$uc$  was created by LPO and transferred to *script\_RP\_index* in a *postMessage* by a script of LPO running in  $b$  (either *script\_LPO\_ld*, *postMessage* sent in Line 82 of Algorithm 14 or *script\_LPO\_cif*, *postMessage* sent in Line 66 of Algorithm 13) loaded into a document with the origin  $\langle \text{dom}(\text{LPO}), \mathbf{S} \rangle$ .*
3.  *$ia$  contains the origin  $o = \langle d_r, \mathbf{S} \rangle$ .*
4.  *$k_u$  is not known to any atomic DY process except for  $b$ , as long as  $b$  is not fully corrupted.*
5.  *$uc$  is issued for an identity  $i \in \text{ownerOfID}^{-1}(b)$ .*

*Proof.* As we know that the message is sent from the origin  $\langle d_r, \mathbf{S} \rangle$ , we know that the script *script\_RP\_index* was loaded over an HTTPS origin (see Lemma 3). Its script state cannot be manipulated by scripts loaded from a different origin (see Algorithms 3 and 6).

The only transitions of the script *script\_RP\_index* which can send out a request to  $r$  are the ones starting out from state *scriptstate.q* = **sendCAP** (Line 80 of Algorithm 15). These transitions take the CAP  $c$  from *scriptstate.cap*. The only transitions before which could have written something into this place in the *scriptstate* are the ones where *scriptstate.q* = **default** (Line 25) when handling a *postMessage* from origin LPO (we can overapproximate here by ignoring all other side restrictions of this transition, e.g. having *scriptstate.dialogRunning* =  $\top$ ). This means that the CAP  $c$  was sent by a script with origin LPO. Since origin LPO is also an HTTPS origin, the script must have been sent by LPO (Lemma 2).

The *postMessage* that was received by *script\_RP\_index* is checked to be a sequence with the first element being **login** or **response**. Such *postMessages* are issued by LPO only in Line 80 of Algorithm 13 (*script\_LPO\_cif*) and in Line 64 of Algorithm 14 (*script\_LPO\_ld*). In both cases,  $ia$  is signed using the private key  $k_u$  that is taken from the respective script's state. This element of the script's state is only written to once, and with a freshly chosen nonce (Line 68 of Algorithm 13 and Line 52 of Algorithm 14, respectively). In both cases, starting in the script state **requestUC**, an XHR to LPO is sent to have  $\text{pub}(k_u)$  signed by LPO. From the response to this request,  $uc$  is extracted. The request is always sent over HTTPS to LPO. Lemma 1, in particular Property (4), applies. Therefore, we see that  $uc$  was actually sent by LPO.

Looking at the LPO definition (Line 60 of Algorithm 10) we see that LPO only sends out freshly created  $uc$ 's. LPO only issues valid UCs (if any). Once returned to *script\_LPO\_cif* or *script\_LPO\_ld*, the UC is combined with an IA and sent to *script\_RP\_index* (which is

determined by the sender of the initial CAP request and its origin). This script sends the CAP to  $r$ . Thus, the CAP that is sent is always valid, which proves (1). Further, the UC was always created by LPO, proving (2).

Property (3) follows immediately from the above observations (i.e.,  $script\_LPO\_cif$  or  $script\_LPO\_ld$  were loaded over HTTPS and are used to create the CAP) and Lemma 8.

To prove (4), we observe that the key  $k_u$  is always chosen freshly and that it is stored only in the script's state. It is not sent to any party, not even LPO. The key therefore cannot leak as long as the browser is not fullycorrupted (if it becomes closecorrupted, the key is removed together with the document's state). This proves (4).

Property (5) follows immediately with the observations in the proof of Property (2) and Lemma 7.  $\square$

**Lemma 10.** *If in a run  $\rho$  of  $\mathcal{BID}$  an IA  $ia$  for an origin  $\langle d_r, \mathcal{S} \rangle$  where  $d_r \in \text{dom}(\text{RP})$  is signed in the scripts  $script\_LPO\_ld$  or  $script\_LPO\_cif$  in an honest browser  $b$ , and these scripts were loaded over HTTPS from LPO, then at most  $b$  and the RP  $r = \text{dom}^{-1}(d_r)$  know  $ia$ .*

*Proof.* The scripts  $script\_LPO\_ld$  and  $script\_LPO\_cif$  send the  $ia$  to the parent or opener window (respectively) using a `postMessage`. For this `postMessage`, the only allowed receiver origin is the same as the origin for which  $ia$  was issued, so in our case  $\langle d_r, \mathcal{S} \rangle$  (see proof for Lemma 9 Property (3)). The script  $script\_RP\_index$ , which thus must be the receiver, sends the complete CAP (containing  $ia$ ) to RP using HTTPS. The RP discards the CAP after checking it. The CAP and especially  $ia$  therefore cannot leak.  $\square$

**Lemma 11.** *In a run  $\rho$  of  $\mathcal{BID}$ , if LPO creates a message containing a UC  $uc$  for an identity  $i$  of a browser  $b$ , then there is no state in the run  $\rho$  where  $b$  is honest and attacker knows the private key  $k_u$  corresponding to the public key  $\text{pub}(k_u)$  that was signed in  $uc$ .*

*Proof.* First, it is easy to see that the Lines 60–63 in Algorithm 10 have to be used in the transition to create  $uc$ : At no other point in the definition of LPO  $uc$  is created or emitted. From Line 54 and following it is easy to see that a request of the following form has to be sent to LPO in order to create  $uc$ :

$$\begin{aligned} req_{uc} = \text{enc}_a(\langle \langle \text{HTTPReq}, n_1, \text{POST}, \text{dom}(\text{LPO}), /certreq, \langle \rangle, \\ \langle [\text{Cookie} : [\text{browserid\_state} : \text{sessionid}], \dots ] \rangle, \\ \langle i, \text{pub}(k_u), \text{xsrftoken} \rangle, k'' \rangle, \text{pub}(\text{key}(\text{dom}(\text{LPO}))) \rangle) \end{aligned} \quad (20)$$

We can see that this message is encrypted with  $\text{pub}(\text{key}(\text{dom}(\text{LPO})))$  and thus the attacker cannot decrypt it. There are now two cases:

- **The attacker knows  $k''$ :** In this case, we can see with Lemma 1 that no honest browser has created  $req_{uc}$ . As RP, LPO, and dishonest browsers do not emit requests in general, only the attacker can have created this request. For this, he needs to know  $\text{xsrftoken}$  and  $\text{sessionid}$ .

The attacker could use a  $\text{sessionid}$  value that was first issued to a browser that was honest when  $req_{uc}$  was created or to some other party (a dishonest browser or himself).

The first case can be ruled out, as the attacker cannot know the  $\text{sessionid}$  value (Lemma 4).

In the second case, he cannot create a message that leads to the authentication of the session himself (which would require knowledge of the secret for identity  $i$ ) and he cannot force the owner browser of  $i$  to authenticate the session (Lemma 7).

- **The attacker does not know  $k''$ :** In this case, the request was not created by the attacker. As above, RP, LPO and dishonest browsers do not create requests. Thus, this request was created by an honest browser (“honest” in the state when  $req_{uc}$  was created) and an honest script in that browser (with a dishonest script, the attacker would need to know  $xsrftoken$ , which he does not according to Lemma 5 and Lemma 4). If an honest script, i.e.,  $script\_LPO\_cif$  or  $script\_LPO\_ld$ , is used, the attacker does not learn  $k_u$  (Lemma 9 Property (4)).

As we can see, in both cases the attacker does not learn  $k_u$ , which proves the statement.  $\square$

**Lemma 12.** *If in a run  $\rho$  of  $\mathcal{BID}$  a browser  $b$  created the request  $req_{cap}$  defined in (17), then (i)  $req_{cap}$  was sent from the script  $script\_RP\_index$  that was loaded over an HTTPS origin from  $r$  while the browser was honest or (ii)  $req_{cap}$  was encrypted by the attacker script while the browser was honest and the attacker knows the CAP  $c$  and the symmetric key  $k$ .*

*Proof.* We can see that if the browser is dishonest, it did encrypt  $req_{cap}$  while it was still honest: With assumption (\*\*\*) we can see that dishonest browsers only send their state to the attacker. Thus, the encrypted message must have been in the state of the browser before corruption. So for both (i) and (ii) we know that the browser was honest.

In an honest browser, the browser itself can create encrypted requests (when an HTTPS request is sent) and scripts can create encrypted requests (by assembling and encrypting the message in the script relation).

In the former case (HTTPS request), which corresponds to statement (i) of the lemma, we see by Lemma 3 that the script that initiated  $req_{cap}$  was actually loaded from  $r$  using HTTPS ( $r$  is the owner of  $d_r$ ), and that it was not altered by any other party.

In the latter case (script encrypted request), which corresponds to statement (ii) in the lemma, we see that the honest scripts do not encrypt messages and thus, the attacker script is the only script that can do so. To do so, the script needs to know every component of  $req_{cap}$  before the encryption, in particular  $k$  and  $c$ . These have been sent to the attacker before the encryption according to (\*\*\*). Thus, the attacker must know  $k$  and  $c$  before.  $\square$

Let  $m$  be the message that was passed to **attacker** leading to  $s_j$  for some addresses  $x$  and  $y$  (with  $s_j$  as defined in (\*)). That is:

$$s_{j-1} \xrightarrow{(x:y:m) \rightarrow \text{attacker}} s_j .$$

By our assumption, we know that  $\langle n, i \rangle \notin d_{N\text{attacker}}(S_{j-1}(\text{attacker}))$  and that  $\langle n, i \rangle \in d_{N\text{attacker}}(S_{j-1}(\text{attacker}), m)$ .

We now distinguish two cases: (i) The attacker does not know  $k$  in  $s_j$  (i.e., cannot derive  $k$  in state  $s_j$ ). (ii) The attacker can derive  $k$  in  $s_j$ . In both cases we lead (\*) to a contradiction.

### I.1.1 The attacker does not know $k$ in $s_j$

We now assume that  $k \notin d_{N\text{attacker}}(S_j(\text{attacker}))$ , i.e., the attacker does not know  $k$  in  $s_j$ . In particular, we have that  $k \notin d_{N\text{attacker}}(S_{j-1}(\text{attacker}))$ .

We distinguish between the kind of atomic processes that potentially have created  $req_{cap}$ . In all cases, we arrive at a contradiction.

- **The browser that owns  $i$**  created  $req_{cap}$ : By Lemma 12 it follows that the browser was honest when encrypting  $req_{cap}$ , and  $req_{cap}$  was initiated by  $script\_RP\_index$ , which was delivered over HTTPS from  $r$ . Note that we can rule out case (ii) in the lemma, as the attacker does not know  $k$ .

This script initiated  $req_{cap}$  and it is easy to see that this script (or no script at all) receives the corresponding response: From the browser definition, we see that XHR responses are delivered to the document with the same nonce as the document that initiated the request (Line 43 in Algorithm 7). Other documents have no access to the data from this document, except for same-origin documents (this is ensured by the `Clean` function that is used in Line 3 of Algorithm 6 and by the `GETWINDOW` function (Algorithm 3) that determines the windows which can be manipulated by other scripts). However, other same-origin documents can only contain the script  $script\_RP\_index$  (this is the only script that RP sends, and with Lemma 2 we see that other same-origin documents cannot have been sent by the attacker). Other manipulations to the window of the document (e.g., navigating the window away) change the active document in the window (Algorithm 7) and could only prevent the script from receiving the response.

From Algorithm 15 it is easy to see that after  $\langle n, i \rangle$  is delivered back to  $script\_RP\_index$  after  $resp_{cap}$  was received, nothing happens with  $\langle n, i \rangle$ : If the browser is uncorrupted, only same-origin scripts have access to it (as shown above), but there are no scripts which use the information. The information can therefore not leak to the attacker. If the browser is closecorrupted before receiving  $resp_{cap}$ , the attacker cannot derive  $\langle n, i \rangle$  from its information, as the decryption key is lost. If the browser is closecorrupted after receiving  $resp_{cap}$ , by definition of close-corruption,  $\langle n, i \rangle$  is removed from the browser's state before the browser can be controlled by the adversary. By the assumption in (\*), the browser cannot be fullycorrupted at any point in the run. Hence, in contradiction to (\*), the attacker cannot obtain  $\langle n, i \rangle$ .

- **A browser that does not own  $i$**  created  $req_{cap}$ : In this case, it still holds that the browser was honest when encrypting  $req_{cap}$  and the script  $script\_RP\_index$  created the request and was loaded over HTTPS (Lemma 12). With Lemma 3 and Lemma 9, in particular Properties (2) and (5), we see that the RP script only initiates HTTPS requests containing CAPs that have been created by LPO and for an identity of the browser. This is in contradiction to the fact that  $i$  is not owned by the browser but  $req_{cap}$  contains a CAP for  $i$ . Hence,  $req_{cap}$  cannot have been created by this browser.
- **RPs or LPO** created  $req_{cap}$ : As per their definitions (Algorithms 10 and 11), they do not initiate or create HTTP(S) requests.
- **The attacker process** created  $req_{cap}$ : It is clear that any atomic process that created  $req_{cap}$  needs to know  $k$ . It follows, by our assumption that the attacker cannot derive  $k$ , that the attacker has not created  $req_{cap}$ .

### I.1.2 The attacker knows $k$ in $s_j$

As above, we distinguish between the kind of atomic processes that potentially have created the request  $req_{cap}$ . We will see that the attacker needs to know the CAP  $c$  to learn  $\langle n, i \rangle$ .

- **The browser that owns  $i$**  created  $req_{cap}$ : By our assumption (\*), this browser cannot be fully corrupted in the run. By Lemma 12, it follows in the case (i) that  $script\_RP\_index$  sent the request and that  $k$  cannot be known by the attacker (with Lemma 1, Property (2)) and hence, the browser cannot have created  $req_{cap}$ . In the case (ii) it follows that the attacker needs to know the CAP  $c$  in order to create the request.
- **A browser not owning  $i$**  created  $req_{cap}$ : By Lemma 12, we see that the browser was honest while encrypting  $req_{cap}$  and (i) that  $script\_RP\_index$  sent the request. With Lemma 9 Property (5) we see that the browser cannot have created  $req_{cap}$  because it only creates requests for its own identities. In the case of (ii) we see that, again, the attacker has to know the cap  $c$  in order to create the request.
- **RPs or LPO** created  $req_{cap}$ : As per their definitions (Algorithms 10 and 11), they do not emit HTTP requests.
- **The attacker process** created  $req_{cap}$ : It is clear that any atomic process that created  $req_{cap}$  needs to know  $c$ .

As we can see, the attacker needs to know  $c = \langle uc, ia \rangle$  before he is able to create  $req_{cap}$ . With Lemma 11 we know that the attacker cannot request  $uc$  itself with the identity  $i$ , and thus, he cannot know the key  $k_u$  that was signed in  $uc$ . Neither can any browser other than  $b$  know  $k_u$ , otherwise the attacker could corrupt this browser and learn  $k_u$ . The key  $k_u$  is needed to create  $ia$ , therefore only in  $b$  the identity assertion  $ia$  can be created, and it can only be created by  $script\_LPO\_cif$  or  $script\_LPO\_ld$  (LPO checks the origin of the request for  $uc$ , and only the script that sends  $uc$  knows  $k_u$ ). With Lemma 10 we see that the attacker cannot learn  $ia$ .

Hence we can see that the attacker cannot know the CAP  $c$  that he needs in order to create  $req_{cap}$ . In particular, he cannot know the key  $k$  that was used to encrypt the response  $resp_{cap}$ , in contradiction to the assumption that the attacker knows  $k$ .  $\square$

## I.2 Condition B

We assume that Condition B is not satisfied and prove that this leads to a contradiction. That is, we make the following assumption: There is a run  $\rho$  of  $\mathcal{BID}$ , a state  $(S_j, E_j)$  in  $\rho$ , an  $r \in \mathbf{RP}$ , an RP service token of the form  $\langle n, i \rangle$  recorded in  $r$  in the state  $S_j(r)$ , the request corresponding to  $\langle n, i \rangle$  was sent by some  $b \in \mathbf{B}$  which is honest in  $S_j$  and  $b$  does not own  $i$ .

Again, without loss of generality, we may assume that  $\rho$  satisfies (\*\*) and (\*\*\*) as above.

As above, the request  $req_{cap}$  corresponding to  $\langle n, i \rangle$  is of the following form

$$req_{cap} = \text{enc}_a(\langle \langle \text{HTTPReq}, n_{cap}, \text{POST}, d_r, /, \langle \rangle, [\text{Origin} : \langle d_r, \mathbf{S} \rangle], c \rangle, k \rangle, \text{pub}(k_r)) \quad (21)$$

with  $d_r$ ,  $n_{cap}$ ,  $\text{pub}(k_r)$ ,  $k$ ,  $c$  as before.

By Lemma 3, this request was actually initiated by a script of  $r$ , which can only be the script  $script\_RP\_index$ . Lemma 9 says that for any such request,  $b$  must be the owner of  $i$  which is a contradiction to the assumption.  $\square$

## References

- [1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001)*, pages 104–115. ACM Press, 2001.

- [2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 290–304. IEEE Computer Society, 2010.
- [3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In V. Shmatikov, editor, *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, pages 1–10. ACM, 2008.
- [4] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.
- [5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffei. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In D. A. Basin and J. C. Mitchell, editors, *Principles of Security and Trust - Second International Conference, POST 2013*, volume 7796 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2013.
- [6] C. Bansal, K. Bhargavan, and S. Maffei. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 247–262. IEEE Computer Society, 2012.
- [7] J. Bellamy-McIntyre, C. Luterroth, and G. Weber. OpenID and the Enterprise: A Model-Based Analysis of Single Sign-On Authentication. In *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2011*, pages 129–138. IEEE Computer Society, 2011.
- [8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001.
- [9] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 11–11. USENIX Association, 2010.
- [10] E. Börger, A. Cisternino, and V. Gervasi. Contribution to a Rigorous Analysis of Web Application Frameworks. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012*, volume 7321 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012.
- [11] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.
- [12] Cross-Origin Resource Sharing - W3C Recommendation 29 January 2013. <http://www.w3.org/TR/2013/CR-cors-20130129/>.
- [13] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

- [14] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 2014. To appear.
- [15] T. Groß. Security Analysis of the SAML Single Sign-on Browser/Artifact Profile. In *19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 298–307. IEEE Computer Society, 2003.
- [16] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. The emperor’s new apis: On the (in)secure usage of new client side primitives. In *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP), 2010*, 2010.
- [17] S. M. Hansen, J. Skriver, and H. R. Nielson. Using static analysis to validate the SAML single sign-on protocol. In C. Meadows, editor, *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005*, pages 27–40. ACM, 2005.
- [18] HTML5, W3C Candidate Recommendation. Dec. 17, 2012.
- [19] HTML5 Web Messaging, W3C Candidate Recommendation. May 1, 2012.
- [20] D. Jackson. Alloy: A new technology for software modelling. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, page 20. Springer, 2002.
- [21] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In P. Ning, S. D. C. di Vimercati, and P. F. Syver-son, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007*, pages 58–71. ACM, 2007.
- [22] F. Kerschbaum. Simple cross-site attack prevention. In *Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.
- [23] A. Kumar. Using automated model analysis for reasoning about security of web protocols. In R. H. Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012*, pages 289–298. ACM, 2012.
- [24] Mozilla Identity Team. Persona. Mozilla Developer Network. Last visited May 1, 2013. <https://developer.mozilla.org/en/docs/persona>.
- [25] OpenID Foundation website. <http://openid.net>.
- [26] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.
- [27] P. Sovis, F. Kohlar, and J. Schwenk. Security Analysis of OpenID. In *Sicherheit*, volume 170 of *LNI*, pages 329–340. GI, 2010.

- [28] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security, CCS'12*, pages 378–390. ACM, 2012.
- [29] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 31(4):465–483, 2012.
- [30] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy (S&P 2012), 21-23 May 2012, San Francisco, California, USA*, pages 365–379. IEEE Computer Society, 2012.
- [31] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *32nd IEEE Symposium on Security and Privacy, S&P 2011*, pages 465–480. IEEE Computer Society, 2011.
- [32] Web Storage - W3C Recommendation 30 July 2013. <http://www.w3.org/TR/2013/REC-webstorage-20130730/>.
- [33] whatwg.org. Fetch. <http://fetch.spec.whatwg.org/>.
- [34] XMLHttpRequest W3C Working Draft. Dec. 6, 2012.