

SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web

Daniel Fett, Ralf Küsters, and Guido Schmitz

University of Trier, Germany
{fett,kuesters,schmitzg}@uni-trier.de

Abstract. Single sign-on (SSO) systems, such as OpenID and OAuth, allow web sites, so-called relying parties (RPs), to delegate user authentication to identity providers (IdPs), such as Facebook or Google. These systems are very popular, as they provide a convenient means for users to log in at RPs and move much of the burden of user authentication from RPs to IdPs.

There is, however, a downside to current systems, as they do not respect users' privacy: IdPs learn at which RP a user logs in. With one exception, namely Mozilla's BrowserID system (a.k.a. Mozilla Persona), current SSO systems were not even designed with user privacy in mind. Unfortunately, recently discovered attacks, which exploit design flaws of BrowserID, show that BrowserID does not provide user privacy either.

In this paper, we therefore propose the first privacy-respecting SSO system for the web, called SPRESSO (for Secure Privacy-REspecting Single Sign-On). The system is easy to use, decentralized, and platform independent. It is based solely on standard HTML5 and web features and uses no browser extensions, plug-ins, or other executables.

Existing SSO systems and the numerous attacks on such systems illustrate that the design of secure SSO systems is highly non-trivial. We therefore also carry out a formal analysis of SPRESSO based on an expressive model of the web in order to formally prove that SPRESSO enjoys strong authentication and privacy properties.

Table of Contents

| | | |
|------|---|----|
| 1 | Introduction | 4 |
| 2 | Description of SPRESSO | 5 |
| 2.1 | Main Features | 5 |
| 2.2 | Login Flow | 7 |
| 2.3 | Implementation Details | 8 |
| 2.4 | Discussion | 9 |
| 3 | Web Model | 12 |
| 3.1 | Communication Model | 12 |
| 3.2 | Web System | 14 |
| 3.3 | Web Browsers | 14 |
| 4 | Indistinguishability of Web Systems | 15 |
| 5 | Formal Model of SPRESSO | 16 |
| 6 | Privacy of SPRESSO | 17 |
| 7 | Authentication of SPRESSO | 19 |
| 8 | Further Related Work | 19 |
| 9 | Conclusion | 20 |
| | References | 20 |
| A | The Web Model | 21 |
| A.1 | Communication Model | 22 |
| A.2 | Scripting Processes | 25 |
| A.3 | Web System | 25 |
| B | Message and Data Formats | 26 |
| B.1 | Notations | 26 |
| B.2 | URLs | 26 |
| B.3 | Origins | 27 |
| B.4 | Cookies | 27 |
| B.5 | HTTP Messages | 27 |
| B.6 | DNS Messages | 28 |
| B.7 | DNS Servers | 28 |
| C | Detailed Description of the Browser Model | 28 |
| C.1 | Notation and Terminology (Web Browser State) | 28 |
| C.2 | Description of the Web Browser Atomic Process | 31 |
| D | Formal Model of SPRESSO | 38 |
| D.1 | Outline | 38 |
| D.2 | Addresses and Domain Names | 38 |
| D.3 | Keys and Secrets | 38 |
| D.4 | Identities | 39 |
| D.5 | Tags and Identity Assertions | 39 |
| D.6 | Corruption | 39 |
| D.7 | Processes in \mathcal{W} (Overview) | 39 |
| D.8 | SSL Key Mapping | 40 |
| D.9 | Web Attackers | 40 |
| D.10 | Network Attackers | 40 |
| D.11 | Browsers | 40 |
| D.12 | Relying Parties | 40 |
| D.13 | Identity Providers | 43 |
| D.14 | Forwarders | 44 |

| | | | |
|---|------|---|----|
| | D.15 | DNS Servers | 45 |
| | D.16 | SPRESSO Scripts | 45 |
| E | | Formal Security Properties Regarding Authentication | 48 |
| F | | Proof of Theorem 2 | 49 |
| | F.1 | Properties of \mathcal{WS}^{auth} | 49 |
| | F.2 | Property A | 52 |
| | F.3 | Property B | 54 |
| G | | Indistinguishability of Web Systems | 55 |
| H | | Formal Proof of Privacy | 57 |
| | H.1 | Formal Model of SPRESSO for Privacy Analysis | 57 |
| | H.2 | Definition of Equivalent Configurations | 58 |
| | H.3 | Privacy Proof | 62 |

1 Introduction

Web-based Single Sign-On (SSO) systems allow a user to identify herself to a so-called relying party (RP), which provides some service, using an identity that is managed by an identity provider (IdP), such as Facebook or Google. If an RP uses an SSO system, a user does not need a password to log in at the RP. Instead, she is authenticated by the IdP, which exchanges some data with the RP so that the RP is convinced of the user’s identity. When logged in at the IdP already, a user can even log in at the RP by one click without providing any password. This makes SSO systems very attractive for users. These systems are also very convenient for RPs as much of the burden of user authentication, including, for example, the handling of user passwords and lost passwords, is shifted to the IdPs. This is why SSO systems are very popular and widely used on the web. Over the last years, many different SSO systems have been developed, with OpenID [13] (used by Google, Yahoo, AOL, and Wordpress, for example) and OAuth [14] (used by Twitter, Facebook, PayPal, Microsoft, GitHub, and LinkedIn, for example) being the most prominent of such systems; other SSO systems include SAML/Shibboleth, CAS, and WebAuth.

There is, however, a downside to these systems: with one exception, none of the existing SSO systems have been designed to respect users’ privacy. That is, the IdP always knows at which RP the user logs in, and hence, which services the user uses. In fact, exchanging user data between IdPs and RPs directly in every login process is a key part of the protocols in OpenID and OAuth, for example, and thus, IdPs can easily track users.

The first system so far which was designed with the intent to respect users’ privacy was the BrowserID system [19,20], which is a relatively new system developed by Mozilla and is also known by its marketing name *Persona*.

Unfortunately, in [12] severe attacks against BrowserID were discovered, which show that the privacy of BrowserID is completely broken: these attacks allow malicious IdPs and in some versions of the attacks even arbitrary parties to check the login status of users at any RP with little effort (see Section 2.1 for some more details on these attacks). Even worse, these attacks exploit design flaws of BrowserID that, as discussed in [12], cannot be fixed without a major redesign of the system, and essentially require building a new system. As further discussed in Section 2.4, besides the lack of privacy there are also other issues that motivate the design of a new system.

The goal of this work is therefore to design the (first) SSO system which respects users’ privacy in the sense described above, i.e., IdPs (even completely malicious ones) should not be able to track at which RPs users log in. Moreover, the history of SSO systems shows that it is highly non-trivial to design secure SSO systems, not only w.r.t. privacy requirements, but even w.r.t. authentication requirements. Attacks easily go unnoticed and in fact numerous attacks on SSO systems, including attacks on OAuth, OpenID, Google ID, Facebook Connect, SAML, and BrowserID have been uncovered which compromise the security of many services and users at once [4–6, 21, 22, 25–28]. Besides designing and implementing a privacy-respecting SSO system, we therefore also carry out a formal security analysis of the system based on an expressive model of the web infrastructure in order to provide formal security guarantees. More specifically, the contributions of our work are as follows.

Contributions of this Paper. In this work, we propose the system SPRESSO (for Secure Privacy-REspecting Single Sign-On). This is the first SSO system which respects user’s privacy. The system allows users to log in to RPs with their email addresses. A user is authenticated to an RP by the IdP hosting the user’s email address. This is done in such a way that the IdP does not learn at which RP the user wants to log in.

Besides strong authentication and privacy guarantees (see also below), SPRESSO is designed in such a way that it can be used across browsers, platforms, and devices. For this purpose, SPRESSO is based solely on standard HTML5 and web features and uses no browser extensions, plug-ins, or browser-independent executables.

Moreover, as further discussed in Section 2.1, SPRESSO is designed as an open and decentralized system. For example, in contrast to OAuth, SPRESSO does not require any prior coordination or setup between RPs and IdPs: users can log in at any RP with any email address with SPRESSO support.

We formally prove that SPRESSO enjoys strong authentication and privacy properties. Our analysis is based on an expressive Dolev-Yao style model of the web infrastructure [10]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as new technologies, such as web

storage and cross-document messaging (postMessages). JavaScript is modeled in an abstract way by so-called scripting processes which can be sent around and, among others, can create iframes and initiate XMLHttpRequests (XHRs). Browsers may be corrupted dynamically by the adversary.

So far, this web model has been employed to analyze trace-based properties only, namely, authentication properties. In this work, we formulate, for the first time, strong indistinguishability/privacy properties for web applications. Our general definition is not tailored to a specific web application, and hence, should be useful beyond our analysis of SPRESSO. These properties require that an adversary should not be able to distinguish two given systems. In order to formulate these properties we slightly modify and extend the web model.

Finally, we formalize SPRESSO in the web model and formally state and prove strong authentication and privacy properties for SPRESSO. The authentication properties we prove are central to any SSO system, where our formulation of these properties follows the one in [10]. As for the privacy property, we prove that a malicious IdP cannot distinguish whether an honest user logs in at one RP or another. The analysis we carry out in this work is also interesting by itself, as web applications have rarely been analyzed based on an expressive web model (see Section 8).

Structure of this Paper. In Section 2, we describe our system and discuss and motivate design choices. We then, in Section 3, briefly recall the general web model from [10] and explain the modifications and extensions we made. The mentioned strong but general definition of indistinguishability/privacy for web applications is presented in Section 4. In Section 5, we provide the formal model of SPRESSO, based on which we state and analyze privacy and authentication of SPRESSO in Sections 6 and 7, respectively. Further related work is discussed in Section 8. We conclude in Section 9. All details and proofs are available in the appendix. An online demo and the source code of SPRESSO are available at [23].

2 Description of SPRESSO

In this section, we first briefly describe the main features of SPRESSO. We then provide a detailed description of the system in Section 2.2, with further implementation details given in Section 2.3. To provide additional intuition and motivation for the design of SPRESSO, in Section 2.4 we discuss potential attacks against SPRESSO and why they are prevented.

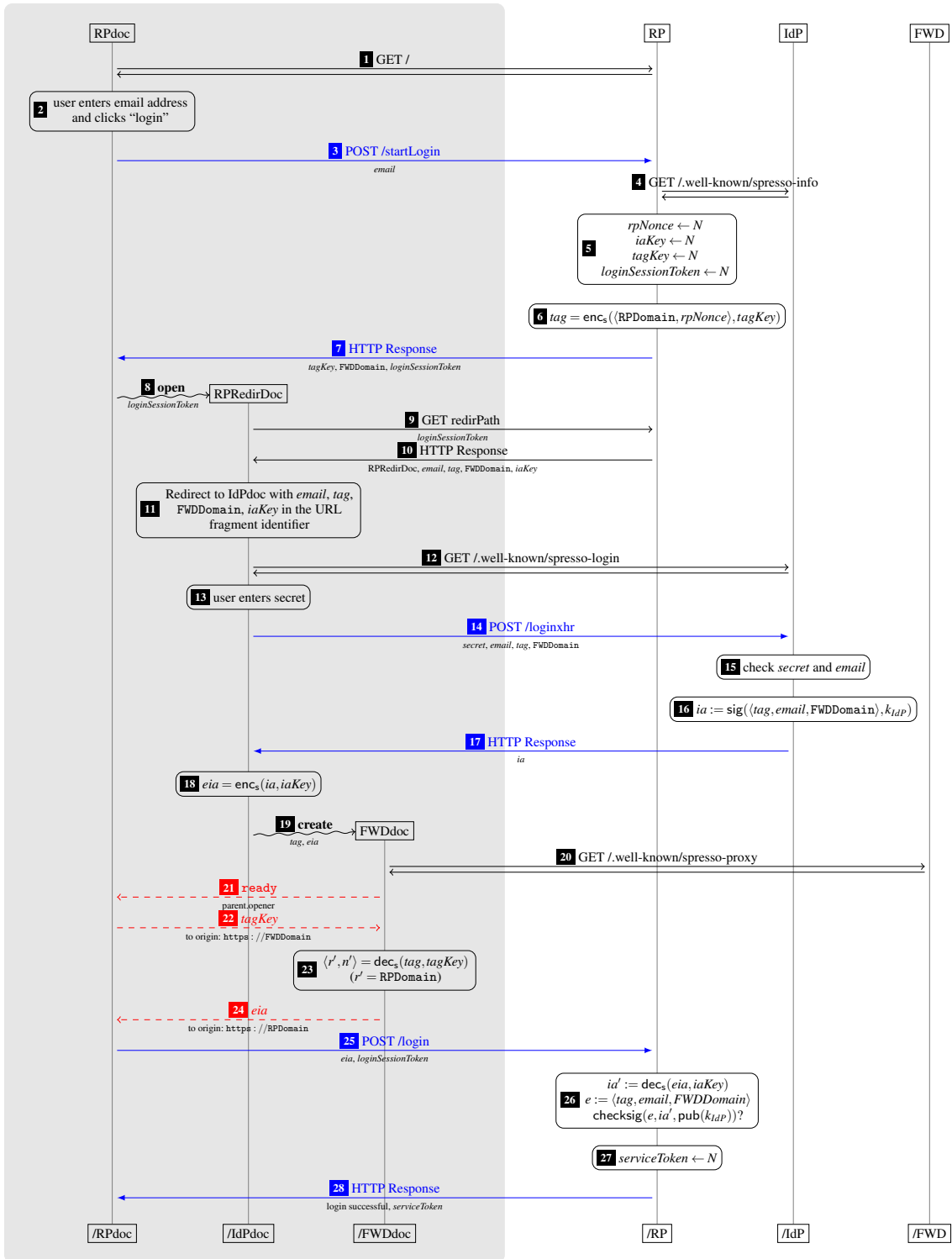
2.1 Main Features

SPRESSO enjoys the following key features:

Strong Authentication and Privacy. SPRESSO is designed to satisfy strong authentication and privacy properties.

Authentication is the most fundamental security property of an SSO system. That is, i) an adversary should not be able to log in to an RP, and hence, use the service of the RP, as an honest user, and ii) an adversary should not be able to log in the browser of an honest user under an adversary’s identity (identity injection). Depending on the service provided by the RP, a violation of ii) could allow the adversary to track an honest user or to obtain user secrets. We note that in the past, attacks on authentication have been found in almost all deployed SSO systems (e.g., OAuth, OpenID, and BrowserID [10, 12, 22, 24, 25, 27, 28]).

While authentication assumes the involved RP and IdP to be honest, privacy is concerned with malicious IdPs. This property requires that (malicious) IdPs should not be able to track at which RPs specific users log in. As already mentioned in the introduction, so far, except for BrowserID, no other SSO system was designed to provide privacy. (In fact, exchanging user data between IdPs and RPs directly is a key part of the protocols in OpenID and OAuth, for example, and hence, in such protocols, IdPs can easily track at which RP a user logs in.) However, BrowserID failed to provide privacy: As shown in [12], a subtle attack allowed IdPs (and in some versions of the attack even arbitrary parties) to check the login status of users at any RP. More specifically, by running a malicious JavaScript within the user’s browser, an IdP can, for any RP, check whether the user is logged in at that RP by triggering the (automatic) login process and testing whether a certain iframe is created during this process or not. The (non-)existence of this iframe immediately reveals the user’s login status. Hence, a malicious IdP can track at which RP a user is logged in. As we discuss in [12], this could not be fixed without a major redesign of BrowserID. Our work could be considered such



→ HTTPS messages, → XHRs (over HTTPS), - -> postMessages, ~> browser commands

Fig. 1. SPRESSO Login Flow.

a major redesign. While SPRESSO shares some basic concepts with BrowserID, SPRESSO is, however, not based on BrowserID, but a new system built from scratch (see the discussion in Section 2.4).

The above shows that the design of a secure SSO system is non-trivial and that attacks are very easy to overlook. As already mentioned in the introduction, we therefore not only designed and implemented SPRESSO to meet strong authentication and privacy properties, but also perform a formal analysis of SPRESSO in an expressive model of the web infrastructure in order to show that SPRESSO in fact meets these properties.

An Open and Decentralized System. We created SPRESSO as a decentralized, open system. In SPRESSO, users are identified by their email addresses, and email providers certify the users' authenticity. Compared to OpenID, users do not need to learn a new, complicated identifier — an approach similar to that of BrowserID. But unlike in BrowserID, there is no central authority in SPRESSO (see also the discussion in Section 2.4). In contrast to OAuth, SPRESSO does not require any prior coordination or setup between RPs and IdPs: Users can log in at any RP with any email address with SPRESSO support. For email addresses lacking SPRESSO support, a seamless fallback can be provided, as discussed later.

Adherence to Web Standards. SPRESSO is based solely on standard HTML5 and web features and uses no browser extensions, plug-ins, or other client-side executables. This guarantees that SPRESSO can be used across browsers, platforms and devices, including both desktop computers and mobile platforms, without installing any software (besides a browser). Note that on smartphones, for example, browsers usually do not support extensions or plug-ins.

2.2 Login Flow

We now explain SPRESSO by a typical login flow in the system. SPRESSO knows three distinct types of parties: relying parties (RPs), i.e., web sites where a user wishes to log in, identity providers (IdPs), providing to RPs a proof that the user owns an email address (identity), and forwarders (FWDs), who forward messages from IdPs to RPs within the browser. We start with a brief overview of the login flow and then present the flow in detail.

Overview. On a high level, the login flow consists of the following steps: First, on the RP web site, the user enters her email address. RP then creates what we call a *tag* by encrypting its own domain name and a nonce with a freshly generated symmetric key. This tag along with the user's email address is then forwarded to the IdP. Due to the privacy requirement, this is done via the user's browser in such a way that the IdP does not learn from which RP this data was received. Note also that the tag contains RP's domain in encrypted form only. The IdP then signs the tag and the user's email address (provided that the user is logged in at the IdP, otherwise the user first has to log in). This signature is called the *identity assertion (IA)*. The IA is then transferred to the RP (again via the user's browser), which checks the signature and consistency of the data signed and then considers the user with the given email address to be logged in. We note that passing the IA to the RP is done using an FWD (the RP determines which one is used) as it is important that the IA is delivered to the correct RP (RP document). The IdP cannot ensure this, because, again due to the privacy requirements, IdP is not supposed to know the intended RP.

Detailed Flow. We now take a detailed look at the SPRESSO login flow. We refer to the steps of the protocol as depicted in Figure 1. We use the names RP, IdP, and FWD for the servers of the respective parties. We use RPdoc, RPRedirDoc, IdPdoc, and FWDdoc as names for HTML documents delivered by the respective parties. The login flow involves the servers RP, IdP, and FWD as well as the user's browser (gray background), in which different windows/iframes are created: first, the window containing RPdoc (which is present from the beginning), second, the login dialog created by RPdoc (initially containing RPRedirDoc and later IdPdoc), and third, an iframe inside the login dialog where the document FWDdoc from FWD is loaded.

As the first step in the protocol, the user opens the login page at RP [1]. The actual login then starts when the user enters her email address [2]. RPdoc sends this address in a POST request to RP [3]. RP identifies the IdP (from the domain in the email address) and retrieves a *support document* from IdP [4]. This document is retrieved from a fixed URL `https://IdPdomain/.well-known/spresso-info` and contains a public (signature verification) key of the IdP. RP now selects new nonces/symmetric keys *rpNonce*, *iaKey*, *tagKey*, and *loginSessionToken* [5] and creates the tag *tag* by encrypting RP's domain `RPDomain` and the nonce *rpNonce* under *tagKey* [6]. Using standard Dolev-Yao notation

(see also Section 3), we denote this term by

$$tag := enc_s(\langle RPDomain, rpNonce \rangle, tagKey).$$

RP further selects an FWD (e.g., a fixed one from its settings). Now, RP stores tag , $iaKey$, the FWD domain, and the email address in its session data store under the session key $loginSessionToken$ and sends $tagKey$, $FWDDomain$, and $loginSessionToken$ as response to the POST request by RPdoc [7].

RPdoc now opens the login dialog. Ultimately, this window contains the login dialog from IdP (IdPdoc) so that the user can log in to IdP (if not logged in already). However, to preserve the user’s privacy (see the discussion in Section 2.4), RPdoc does not launch the dialog with the URL of IdPdoc immediately. Instead, RPdoc opens the login dialog with the URL of RPRedirDoc and attaches the $loginSessionToken$ [8]. RPRedirDoc is loaded from RP [9] and [10] and redirects the login dialog to IdPdoc [11] and [12], passing the user’s email address, the tag, the FWD domain, and the $iaKey$ from RP, as stored under the session key $loginSessionToken$, to IdPdoc.¹

After the browser loaded IdPdoc from IdP, the user enters her password² matching her email address [13]. The password, the email address, the tag, and the FWD domain are now sent to IdP [14]. After IdP verified the user credentials [15], it creates the identity assertion as the signature

$$ia := sig(\langle tag, email, FWDDomain \rangle, k_{IdP})$$

using its private signing key k_{IdP} [16] and then returns ia to IdPdoc [17]. We note that ia contains the signature only, not the data that was signed.

To avoid that the FWD learns the IA (we discuss this further in Section 2.4), IdPdoc now encrypts the IA using the $iaKey$ [18]:

$$eia := enc_s(ia, iaKey).$$

Then, IdPdoc opens an iframe with the URL of FWDdoc, passing the tag and the encrypted IA to FWDdoc. After the iframe is loaded [20], FWDdoc sends a $postMessage$ ³ to its parent’s opener window, which is RPdoc [21]. This $postMessage$ with the sole content “ready” triggers RPdoc to send the $tagKey$ to FWDdoc, where in the $postMessage$ the origin⁴ of FWD with HTTPS is declared to be the only allowed receiver of this message [22]. FWDdoc uses the key to decrypt the tag and thereby learns the intended receiver (RP) of the IA [23]. As its last action, FWD forwards the encrypted IA eia via $postMessage$ to RPdoc (using RP’s HTTPS origin as the only allowed receiver) [24].

RPdoc receives eia and sends it along with the $loginSessionToken$ to RP [25]. RP then decrypts eia , retrieves ia' and checks whether ia' is a valid signature for $\langle tag, email, FWDDomain \rangle$ under the verification key $pub(k_{IdP})$ of the IdP, where tag , $email$, and $FWDDomain$ are taken from the session data identified by $loginSessionToken$ [26].

Now, the user identified by the email address is logged in. The mechanism that is used to persist this logged-in state (if any) at this point is out of the scope of SPRESSO. In our analysis, as a model for a standard session-based login, we assume that RP creates a session for the user’s browser, identified by some freshly chosen token (the RP service token) [27] and sends this token to the browser [28].

2.3 Implementation Details

We developed a proof-of-concept implementation of SPRESSO in about 700 lines of JavaScript and HTML code. It contains all presented features of SPRESSO itself and a typical IdP. The implementation (source code and online demo) is available at [23]. Our model presented in Section 5 closely follows this implementation.

The three servers (RP, IdP and FWD) are written in JavaScript and are based on node.js and its built-in *crypto* API. On the client-side we use the Web Cryptography API. For encryption we employ AES-256 in GCM mode to provide authenticity. Signatures are created/verified using RSA-SHA256.

¹This data is passed to IdPdoc in the fragment identifier of the URL (a.k.a. *hash*), and therefore, it is not necessarily sent to IdP.

²In fact, the IdP can as well offer any other form of authentication, e.g., TLS client authentication or two-factor authentication.

³ $postMessages$ are messages that are sent between different windows in one browser.

⁴An origin is defined by a domain name plus the information whether the connection to this domain is via HTTP or HTTPS.

2.4 Discussion

In order to provide more intuition and motivation for the design of SPRESSO, and in particular its security and privacy properties, we first informally discuss some potential attacks on our system and what measures we took when designing and implementing SPRESSO to prevent these attacks. These attacks also illustrate the complexity and difficulty of designing a secure and privacy-respecting web-based SSO system. In Sections 6 and 7, we formally prove that SPRESSO provides strong authentication and privacy properties in a detailed model of the web infrastructure. We also discuss other aspects of SPRESSO, including usability and performance. We conclude this section with a comparison of SPRESSO and BrowserID.

Malicious RP: Impersonation Attack. An attacker could try to launch a man in the middle attack against SPRESSO by playing the role of an RP (RP server and RPdoc) to the user. Such an attacker would run a malicious server at his RP domain, say, RP_a , and also deliver a malicious script (instead of the honest RPdoc script) to the user's browser. Now assume that the user wants to log in with her email address at RP_a and is logged in at the IdP corresponding to the email address already. Then, the attacker (outside of the user's browser) could first initiate the login process at RP_b using the user's email address. The attacker's RP could then create a tag of the form $enc_s(\langle RP_b, rpNonce \rangle, tagKey)$ using the domain of an honest RP RP_b , instead of RP_a . The IdP would hence create an IA for this tag and the user's email address and deliver this IA to the user's browser. If this IA were now indeed be delivered to the attacker's RP window (which is running a malicious RPdoc script), the attacker could use the IA to finish the log in process at RP_b (and obtain the service token from RP_b), and thus, log in at RP_b as the honest user.

However, assuming that FWD is honest (see below for a discussion of malicious FWDs), FWD prevents this kind of attack: FWD forwards the (encrypted) IA via a `postMessage` only to the domain listed in the tag (so, in this case, RP_b), which in the attack above is not the domain of the document loaded in the attacker's RP window (RP_a). The IA is therefore not transmitted to the attacker. The same applies when the attacker tries to navigate the RP window to its own domain, i.e., to RP_a , before Step [24]. Our formal analysis presented in the following sections indeed proves that such attacks are excluded in SPRESSO. We note that in order to make sure that the `postMessage` is delivered to the correct RP window (technically, a window with the expected origin), FWD uses a standard feature of the `postMessage` mechanism which allows to specify the origin of the intended recipient of a `postMessage`.

Malicious IdP. A malicious IdP could try to log the user in under an identity that is not her own. An attack of this kind on BrowserID was shown in [10]. However, in SPRESSO, the IdP cannot select or alter the identity with which the user is logged in. Instead, the identity is fixed by RP after Step [6] and checked in Step [26]. Again, our formal analysis shows that such attacks are indeed not possible in SPRESSO.

The IdP could try to undermine the user's privacy by trying to find out which RP requests the IA. However, in SPRESSO, the IdP cannot gather such information: From the information available to it (*email, tag, FWDDomain* plus any information it can gather from the browser's state), it cannot infer the RP.⁵ It could further try to correlate the sources and times of HTTPS requests for the support document with user logins. To minimize this side channel, we suggest caching the support document at each RP and automatic refreshing of this cache (e.g., an RP could cache the document for 48 hours and after that period automatically refresh the cache). Additionally, RPs should use the Tor network (or similar means) when retrieving the support document in order to hide their IP addresses. Assuming that support documents have been obtained from IdPs independently of specific login requests by users, our formal analysis shows that SPRESSO in fact enjoys a very strong privacy property (see Sections 4 and 6).

In BrowserID, malicious IdPs (in fact, any party who can run malicious scripts in the user's browser) can check the presence or absence of certain iframes in the login process, leading to the privacy break mentioned earlier. Again, our formal analysis implies that this is not possible for SPRESSO.

Malicious FWD. A malicious FWD could cooperate with or act as a malicious RP and thereby enable the man in the middle attack discussed above, undermining the authentication guarantees of the system. Also, a malicious FWD could collaborate with a malicious IdP and send information about the RP to the IdP, and hence, undermine privacy.

⁵If only a few RPs use a specific FWD, *FWDDomain* would reveal some information. However, this is easy to avoid in practice: the set of FWDs all (or many) RPs trust should be big enough and RPs could randomly choose one of these FWDs for every login process.

Therefore, for our system to provide authentication and privacy, we require that FWDs behave honestly. Below we discuss ways to force FWD to behave honestly. We suspect that there is no way to avoid the use of FWDs or other honest components in a practical SSO system which is supposed to provide not only authentication but also privacy: In our system, after Step [17] of the flow, IdPdoc must return the IA to the RP. There are two constraints: First, the IA should only be forwarded to a document that in fact is RP's document. Otherwise, it could be misused to log in at RP under the user's identity by any other party, which would break authentication. Second, RP's identity should not be revealed to IdP, which is necessary for privacy. Currently, there is no browser mechanism to securely forward the IA to RP without disclosing RP's identity to IdP (but see below).

Enforcing Honest FWDs. Before we discuss existing and upcoming technologies to enforce honest behavior of FWDs, we first note that in SPRESSO, an FWD is chosen by the RP to which a user wants to log in. So the RP can choose the FWD it trusts. The RP certainly has a great interest in the trustworthiness of the FWD: As mentioned, a malicious FWD could allow an attacker to log in as an honest user (and hence, misuse RP's service and undermine confidentiality and integrity of the user's data stored at RP), something an RP would definitely want to prevent. Second, we also note that FWD does not learn a user's email address: the IA, which is given to FWD and which contains the user's email address, is encrypted with a symmetric key unknown to FWD.⁶ Therefore, SPRESSO does not provide FWD with information to track at which RP a specific user logs in.⁷

Now, as for *enforcing* honest FWDs, first note that an honest FWD server is supposed to always deliver the same fixed JavaScript to a user's browsers. This JavaScript code is very short (about 50 lines of code). If this code is used, it is not only ensured that FWD preserves authentication and privacy, but also that no tracking data is sent back to the FWD server.

Using current technology, a user could use a browser extension which again would be very simple and which would make sure that in fact only this specific JavaScript is delivered by FWD (upon the respective request). As a result, FWD would be forced to behave honestly, without the user having to trust FWD. Another approach would be an extension that replaces FWD completely, which could also lead to a simplified protocol. In both cases, SPRESSO would provide authentication and privacy without having to trust any FWD. Both solutions have the common problem that they do not work on all platforms, because not on all platforms browsers support extensions. The first solution (i.e., the extension checks only that correct JavaScript is loaded) would at least still work for users on such platforms, albeit with reduced security and privacy guarantees.

A native web technology called *subresource integrity* (SRI)⁸ is currently under development at the W3C. SRI allows a document to create an iframe with an attribute *integrity* that takes a hash value. The browser now would guarantee that the document loaded into the iframe hashes to exactly the given value. So, essentially the creator of the iframe can enforce the iframe to be loaded with a specific document. This would enable SPRESSO to automatically check the integrity of FWDdoc without any extensions.

Referer Header and Privacy. The *Referer* [sic!] header is set by browsers to show which page caused a navigation to another page. It is set by all common browsers. To preserve privacy, when the loading of IdPdoc is initiated by RPdoc, it is important that the Referer header is *not* set, because it would contain RP's domain, and consequently, IdP would be able to read off from the Referer header to which RP the user wants to log in, and hence, privacy would be broken. With HTML5, a special attribute for links in HTML was introduced, which causes the Referer header to be suppressed (`rel="noreferrer"`). However, when such a link is used to open a new window, the new window does not have a handle on the opening window (opener) anymore. But having a handle is essential for SPRESSO, as the `postMessage` in Step [21] is sent to the opener window of IdPdoc. To preserve the opener handle while at the same time hiding the referer, we first open the new window with a redirector document loaded from RP (Step [8]) and then navigate this window to IdPdoc (using a link with the `noreferrer` attribute set and triggered by JavaScript in Step [11]). This causes the

⁶We note that IA is a signature anyway, so typically a signed hash of a message. Hence, for common signature schemes, already from the IA itself FWD is not able to extract the user's email address. In addition, SPRESSO even encrypts the IA to make sure that this is the case no matter which signature scheme is used.

⁷A malicious FWD could try to set cookies and do browser fingerprinting to track the behavior of specific browsers. Still it does not obtain the user's email address.

⁸<http://www.w3.org/TR/SRI/>

Referer header to be cleared, while the opener handle is preserved.⁹ Our formal analysis implies that with this solution indeed privacy is preserved.

Cross-Site Request Forgery. Cross-site request forgery is particularly critical at RP, where it could be used to log a user in under an identity that is not her own. For RP, SPRESSO therefore employs a session token that is not stored in a cookie, but only in the state of the JavaScript, avoiding cross-origin and cross-domain cookie attacks. Additionally, RP checks the Origin header of the login request to make sure that no login can be triggered by a third party (attacker) web page. Our formal analysis implies that cross-site request forgery and related attacks are not possible in SPRESSO.

Phishing. It is important to notice that in SPRESSO the user can verify the location and TLS certificate of IdPdoc's window by checking the location bar of her browser. The user can therefore check where she enters her password, which would not be possible if IdPdoc was loaded in an iframe. Setting strict transport security headers can further help in avoiding phishing attacks.

Tag Length Side Channel. The length of the tag created in Step 6 depends on the length of RPDomain. Since the tag is given to IdP, IdP might try to infer RPDomain from the length of the tag. However, according to RFC 1035, domain names may at most be 253 characters long. Therefore, by appropriate padding (e.g., encrypting always nine 256 Bit plaintext blocks)¹⁰ the length of the tag will not reveal any information about RPDomain.

Performance. SPRESSO uses only standard browser features, employs only symmetric encryption/decryption and signatures, and requires (in a minimal implementation) eight HTTPS requests/responses — all of which pose no significant performance overhead to any modern web application, neither for the browser nor for any of the servers. In our prototypical and unoptimized implementation, a login process takes less than 400 ms plus the time for entering email address and password.

Usability. In SPRESSO, users are identified by their email addresses (an identifier many users easily memorize) and email providers serve as identity providers. Many web applications today already use the email address as the primary identifier along with a password for the specific web site: When a user signs up, a URL with a secret token is sent to the user's email address. The user has to check her emails and click on the URL to confirm that she has control over the email address. She also has to create a password for this web site. SPRESSO could seamlessly be integrated into this sign up scheme and greatly simplify it: If the email provider (IdP) of the user supports SPRESSO, an SPRESSO login flow can be launched directly once the user entered her email address and clicked on the login button, avoiding the need for a new user password and the email confirmation; and if the user is logged in at the IdP already, the user does not even have to enter a password. Otherwise, or if a user has JavaScript disabled, an automatic and seamless fallback to the classical token-based approach is possible (as RP can detect whether the IdP supports SPRESSO in Step 4 of the protocol). In contrast to other login systems, such as Google ID, the user would not even have to decide whether to log in with SPRESSO or not due to the described seamless integration of SPRESSO. Due to the privacy guarantees (which other SSO systems do not have), using SPRESSO would not be disadvantageous for the user as her IdPs cannot track to which RPs the user logs in.

The above illustrates that, using SPRESSO, signing up to a web site is very convenient: The user just enters her email address at the RP's web site and presses the login button (if already logged in at the respective IdP, no password is necessary). Also, with SPRESSO the user is free to use any of her email addresses.

Extendability. SPRESSO could be extended to have the IdP sign (in addition to the email address) further user attributes in the IA, which then might be used by the RP.

Operating FWD. Operating an FWD is very cheap, as the only task is to serve one static file. Any party can act as an FWD. Users and RPs might feel most confident if an FWD is operated by widely trusted non-profit organizations, such as Mozilla or the EFF.

⁹Another option would have been to use a data URI instead of loading the redirector document from RPdoc and to use a Refresh header contained in a meta tag for getting rid of the Referer header. This however showed worse cross-browser compatibility, and the Refresh header lacks standardization.

¹⁰Eight 256 bit blocks are sufficient for all domain names. We need an additional block for *rpNonce*.

Comparison with BrowserID. BrowserID was the first and so far only SSO system designed to provide privacy (IdPs should not be able to tell at which RPs user’s log in). Nonetheless, as already mentioned (see Section 2.1), severe attacks were discovered in [12] which show that the privacy promise of BrowserID is broken: not only IdPs but even other parties can track the login behavior of users. Regaining privacy would have required a major redesign of the system, resulting in essentially a completely new system, as pointed out in [12]. Also, BrowserID has the disadvantage that it relies on a single trusted server (`login.persona.org`) which is quite complex, with several server interactions necessary in every login process, and most importantly, by design, gets full information about the login behavior of users (the user’s email address and the RP at which the user wants to log in).¹¹ Finally, BrowserID is a rather complex SSO system (with at least 64 network and inter-frame messages in a typical login flow¹² compared to only 19 in SPRESSO). This complexity implies that security vulnerability go unnoticed more easily. In fact, several attacks on BrowserID breaking authentication and privacy claims were discovered (see [10, 12]).

This is why we designed and built SPRESSO from scratch, rather than trying to redesign BrowserID. The design of SPRESSO is in fact very different to BrowserID. For example, except for HTTPS and signatures of IdPs, SPRESSO uses only symmetric encryption, whereas in BrowserID, users (user’s browsers) have to create public/private key pairs and IdPs sign the user’s public keys. The entities in SPRESSO are different to those in BrowserID as well, e.g., SPRESSO does not rely on the mentioned single, rather complex, and essentially omniscient trusted party, resulting in a completely different protocol flow. The design of SPRESSO is much slimmer than the one of BrowserID.

3 Web Model

Our formal security analysis of SPRESSO (presented in the next sections) is based on the general Dolev-Yao style web model in [10]. As mentioned in the introduction, we changed some details in this model to facilitate the definition of indistinguishability/privacy properties (see Section 4). In particular, we simplified the handling of nonces and removed non-deterministic choices wherever possible. Also, we added the HTTP *Referer* header and the HTML5 *noreferrer* attribute for links.

Here, we only present a very brief version of the web model. The full model, including our changes, is provided in Appendices A–C.

3.1 Communication Model

The main entities in the communication model are *atomic processes*, which are used to model web browsers, web servers, DNS servers as well as web and network attackers. Each atomic process listens to one or more (IP) addresses. A set of atomic processes forms what is called a *system*. Atomic processes 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 one of the atomic processes that listens to the receiver address of that event. The atomic process can then process the event and output new events, which are added to the pool of events, and so on. More specifically, messages, processes, etc. are defined as follows.

Terms, Messages and Events. As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature. The signature Σ for the terms and messages considered in the web model contains, among others, constants (such as (IP) addresses, ASCII strings, and nonces), sequence and projection symbols, and further function symbols, including those for (a)symmetric encryption/decryption and digital signatures. Messages are defined to be ground terms (terms without variables). For example (see also Section 2.2 where we already use the term notation to describe messages), $\text{pub}(k)$ denotes the public key which belongs to the private key k . To provide another example of a message, in the web model, an HTTP request is represented as a ground term containing a nonce, a method (e.g.,

¹¹In SPRESSO, we require that FWD behaves honestly. In a login process, however, the FWD server needs to provide only a fixed single and very simple JavaScript, no further server interaction is necessary. Also, FWD does not get full information and RP in every login process may choose any FWD it trusts. Moreover, as discussed above, there are means to force FWD to provide the expected JavaScript.

¹²Counting HTTP request and responses as well as `postMessages`, leaving out any user requests for GUI elements or other non-necessary resources.

GET or POST), a domain name, a path, URL parameters, request headers (such as Cookie), and a message body. For instance, an HTTP GET request for the URL <http://example.com/show?p=1> is modeled as the term

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

where headers and body are empty. An HTTPS request for r is of the form $\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

Events are terms of the form $\langle a, f, m \rangle$ where a and f are receiver/sender (IP) addresses, and m is a message, for example, an HTTP(S) message as above or a DNS request/response.

The *equational theory* associated with the signature Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation \equiv on terms. It captures the meaning of the function symbols in Σ . For instance, the equation in the equational theory which captures asymmetric decryption is $\text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) = x$. With this, we have that, for example,

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

i.e., these two terms are equivalent w.r.t. the equational theory.

Atomic Processes, Systems and Runs. Atomic Dolev-Yao processes, systems, and runs of systems are defined as follows.

An *atomic Dolev-Yao (DY) process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where I^p is the set of addresses the process listens to, Z^p is a set of states (formally, terms), $s_0^p \in Z^p$ is an initial state, and R^p is a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. This relation models a computation step of the process, which upon receiving an event in a given state non-deterministically moves to a new state and outputs a set of events. It is required that the events and states in the output can be computed (more formally, derived in the usual Dolev-Yao style) from the current input event and state. We note that in [10] the definition of an atomic process also contained a set of nonces which the process may use. Instead of such a set, we now consider a global sequence of (unused) nonces and new nonces generated by an atomic process are taken from this global sequence.

The so-called *attacker process* is an atomic DY process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process is the maximally powerful DY process. It carries out all attacks any DY process could possibly perform and is parametrized by the set of sender addresses it may use. Attackers may corrupt other DY processes (e.g., a browser).

A *system* is a set of atomic processes. A *configuration* (S, E, N) of this system consists of the current states of all atomic processes in the system (S), the pool of waiting events (E , here formally modeled as a sequence of events; in [10], the pool was modeled as a multiset), and the mentioned sequence of unused nonces (N).

A *run* of a system for an initial sequence of events E^0 is a sequence of configurations, where each configuration (except for the initial one) is obtained by delivering one of the waiting events of the preceding configuration to an atomic process p (which listens to the receiver address of the event), which in turn performs a computation step according to its relation R^p . The initial configuration consists of the initial states of the atomic processes, the sequence E^0 , and an initial infinite sequence of unused nonces.

Scripting Processes. The web model also defines scripting processes, which model client-side scripting technologies, such as JavaScript.

A *scripting process* (or simply, a *script*) is defined similarly to a DY process. It is called by the browser in which it runs. The browser provides it with state information s , and the script then, according to its computation relation, outputs a term s' , which represents the new internal state and some command which is interpreted by the browser (see also below). Again, it is required that a script's output is derivable from its input.

Similarly to an attacker process, the so-called *attacker script* R^{att} may output everything that is derivable from the input.

3.2 Web System

A web system formalizes the web infrastructure and web applications. Formally, a *web system* is a tuple

$$(\mathcal{W}, \mathcal{S}, \text{script}, E^0)$$

with the following components:

- The first component, \mathcal{W} , denotes a system (a set of DY processes as defined above) and contains honest processes, web attacker, and network attacker processes. While a web attacker can listen to and send messages from its own addresses only, a network attacker may listen to and spoof all addresses (and therefore is the maximally powerful attacker). Attackers may corrupt other parties. In the analysis of a concrete web system, we typically have one network attacker only and no web attackers (as they are subsumed by the network attacker), or one or more web attackers but then no network attacker. Honest processes can either be web browsers, web servers, or DNS servers. The modeling of web servers heavily depends on the specific application. The web browser model, which is independent of a specific web application, is presented below.
- The second component, \mathcal{S} , is a finite set of scripts, including the attacker script R^{att} . In a concrete model, such as our SPRESSO model, the set $\mathcal{S} \setminus \{R^{\text{att}}\}$ describes the set of honest scripts used in the web application under consideration while malicious scripts are modeled by the “worst-case” malicious script, R^{att} .
- The third component, script , is an injective mapping from a script in \mathcal{S} to its string representation $\text{script}(s)$ (a constant in Σ) so that it can be part of a messages, e.g., an HTTP response.
- Finally, E^0 is a sequence of events, which always contains an infinite number of events of the form $\langle a, a, \text{TRIGGER} \rangle$ for every IP address a in the web system.

A *run* of the web system is a run of \mathcal{W} initiated by E^0 .

3.3 Web Browsers

We now sketch the model of the web browser, with full details provided in Appendix C. A web browser is modeled as a DY process (I^p, Z^p, R^p, s_0^p) .

An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions are modeled as non-deterministic actions of the web browser. For example, the browser itself non-deterministically follows the links in a web page. User data (i.e., passwords and identities) is stored in the initial state of the browser and is given to a web page when needed, similar to the AutoFill feature in browsers.

Besides the user identities and passwords, the state of a web browser (modeled as a term) contains a tree of open windows and documents, lists of cookies, localStorage and sessionStorage data, a DNS server address, and other data.

In the browser state, the *windows* subterm is the most complex one. It contains a window subterm for every open window (of which there may be many at a time), and inside each window, a list of documents, which represent the history of documents that have been opened in that window, with one of these documents being active, i.e., this document is presented to the user and ready for interaction. A document contains a script loaded from a web server and represents one loaded HTML page. A document also contains a list of windows itself, modeling iframes. Scripts may, for example, navigate or create windows, send XHRs and postMessages, submit forms, set/change cookies, localStorage, and sessionStorage data, and create iframes. When activated, the browser provides a script with all data it has access to, such as a (limited) view on other documents and windows, certain cookies as well as localStorage and sessionStorage.

Figure 2 shows a brief overview of the browser relation R^p which defines how browsers behave. For example, when a TRIGGER message is delivered to the browser, the browser non-deterministically chooses an *action*. If, for instance, this action is 1, then an active document is selected non-deterministically, and its script is triggered. The script (with inputs as outlined above), can now output a command, for example, to follow a hyperlink (HREF). In this case, the browser will follow this link by first creating a new DNS request. Once a response to that DNS request arrives, the actual HTTP request (for the URL defined by the script) will be sent out. After a response to that HTTP request arrives, the browser creates a new document from the contents of the response. Complex navigation and security rules ensure that scripts can only manipulate specific aspects of the browser’s state. Browsers can become corrupted, i.e., be taken

PROCESSING INPUT MESSAGE m

$m = \text{FULLCORRUPT}$: $isCorrupted := \text{FULLCORRUPT}$
 $m = \text{CLOSECORRUPT}$: $isCorrupted := \text{CLOSECORRUPT}$
 $m = \text{TRIGGER}$: non-deterministically choose $action$ from $\{1, 2, 3\}$
 $action = 1$: Call script of some active document.
 Outputs new state and $command$.
 $command = \text{HREF}$: \rightarrow Initiate request
 $command = \text{IFRAME}$: Create subwindow, \rightarrow Initiate request
 $command = \text{FORM}$: \rightarrow Initiate request
 $command = \text{SETSCRIPT}$: Change script in given document.
 $command = \text{SETSCRIPTSTATE}$: Change state of script
 in given document.
 $command = \text{XMLHTTPREQUEST}$: \rightarrow Initiate request
 $command = \text{BACK}$ or FORWARD : Navigate given window.
 $command = \text{CLOSE}$: Close given window.
 $command = \text{POSTMESSAGE}$: Send `postMessage` to specified
 document.
 $action = 2$: \rightarrow Initiate request to some URL in new window
 $action = 3$: \rightarrow Reload some document
 $m = \text{DNS response}$: 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

Fig. 2. The basic structure of the web browser relation R^p with an extract of the most important processing steps, in the case that the browser is not already corrupted.

over by web and network attackers. The browser model comprises two types of corruption: *close-corruption*, modeling that a browser is closed by the user, and hence, certain data is removed (e.g., session cookies and opened windows), before it is taken over by the attacker, and *full corruption*, where no data is removed in advance. Once corrupted, the browser behaves like an attacker process.

4 Indistinguishability of Web Systems

We now define the indistinguishability of web systems. This definition is not tailored towards a specific web application, and hence, is of independent interest.

Our definition follows the idea of trace equivalence in Dolev-Yao models (see, e.g., [9]), which in turn is an abstract version of cryptographic indistinguishability.

Intuitively, two web systems are indistinguishable if the following is true: whenever the attacker performs the same actions in both systems, then the sequence of messages he obtains in both runs look the same from the attacker's point of view, where, as usual in Dolev-Yao models, two sequences are said to "look the same" when they are statically equivalent [1] (see below). More specifically, since, in general, web systems allow for non-deterministic actions (also of honest parties), the sequence of actions of the attacker might induce a set of runs. Then indistinguishability says that for all actions of the attacker and for every run induced by such actions in one system, there exists a run in the other system, induced by the same attacker actions, such that the sequences of messages the attacker obtains in both runs look the same to the attacker.

Defining the actions of attackers in web systems requires care because the attacker can control different components of such a system, but some only partially: A web attacker (unlike a network attacker) controls only part of the network. Also an attacker might control certain servers (web servers and DNS servers) and browsers. Moreover, he might control certain scripts running in honest browsers, namely all attacker scripts R^{att} running in browsers; dishonest browsers are completely controlled by the attacker anyway.

We model a single action of the attacker by what we call a (*web system*) *command*; not to be confused with commands output by a script to the browser. A command is of the form

$$\langle i, j, \tau_{\text{process}}, \text{cmd}_{\text{switch}}, \text{cmd}_{\text{window}}, \tau_{\text{script}}, \text{url} \rangle.$$

The first component $i \in \mathbb{N}$ determines which event from the pool of events is processed. If this event could be delivered to several processes (recall that a network attacker, if present, can listen to all addresses), then j determines the process which actually gets to process the event. Now, there are different cases depending on the process to which the event is delivered and depending on the event itself. We denote the process by p and the event by e : i) If p is corrupted (it is a web attacker, network attacker, some corrupted browser or server), then the new state of this process and its output are determined by the term τ_{process} , i.e., this term is evaluated with the current state of the process and the input e . ii) If p is an honest browser and e is not a trigger message (e.g., a DNS or HTTP(S) response), then the browser processes e as usual (in a deterministic way). iii) If p is an honest browser and e is a trigger message, then there are three actions a browser can (non-deterministically) choose from: open a new window, reload a document, or run a script. The term $\text{cmd}_{\text{switch}} \in \{1, 2, 3\}$ selects one of these actions. If it chooses to open a new window, a document will be loaded from the URL url . In the remaining two cases, $\text{cmd}_{\text{window}}$ determines the window which should be reloaded or in which a script is executed. If a script is executed and this script is the attacker script, then the output of this script is derived (deterministically) by the term τ_{script} , i.e., this term is evaluated with the data provided by the browser. The resulting command, if any, is processed (deterministically) by the browser. If the script to be executed is an honest script (i.e., not R^{att}), then this script is evaluated and the resulting command is processed by the browser. (Note that the script might perform non-deterministic actions.) iv) If p is an honest process (but not a browser), then the process evaluates e as usual. (Again, the computation might be non-deterministic, as honest processes might be non-deterministic.)

We call a finite sequence of commands a *schedule*. Given a web system $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$, a schedule σ induces a set of (finite) runs in the obvious way. We denote this set by $\sigma(\mathcal{WS})$. Intuitively, a schedule models the attacker actions in a run. Note that we consider a very strong attacker. He not only determines the actions of all dishonest processes and all attacker scripts, but also schedules all events, not only events intended for the attacker; clearly, the attacker does not get to see explicitly events not intended for him.

Before we can define indistinguishability of two web systems, we need to, as mentioned above, recall the definition of static equivalence of two messages t_1 and t_2 . We say that the messages t_1 and t_2 are *statically equivalent*, written $t_1 \approx t_2$, if and only if, for all terms $M(x)$ and $N(x)$ which contain one variable x and do not use nonces, we have that $M(t_1) \equiv N(t_1)$ iff $M(t_2) \equiv N(t_2)$. That is, every test performed by the attacker yields the same result for t_1 and t_2 , respectively. For example, if k and k' are nonces, and r and r' are different constants, then

$$\text{enc}_a(\langle r, k' \rangle, \text{pub}(k)) \approx \text{enc}_a(\langle r', k' \rangle, \text{pub}(k)).$$

Intuitively, this is the case because the attacker does not know the private key k .

We also need the following terminology. If $(\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ is a web system and p is an attacker process in \mathcal{W} , then we say that $(\mathcal{W}, \mathcal{S}, \text{script}, E^0, p)$ is a *web system with a distinguished attacker process* p . If ρ is a finite run of this system, we denote by $\rho(p)$ the state of p at the end of this run. In our indistinguishability definition, we will consider the state of the distinguished attacker process only. This is sufficient since the attacker can send all its data to this process.

Now, we are ready to define indistinguishability of web systems in a natural way.

Definition 1. Let \mathcal{WS}_0 and \mathcal{WS}_1 be two web system each with a distinguished attacker process p_0 and p_1 , respectively. We say that these systems are *indistinguishable*, written $\mathcal{WS}_0 \approx \mathcal{WS}_1$, iff for every schedule σ and every $i \in \{0, 1\}$, we have that for every run $\rho \in \sigma(\mathcal{WS}_i)$ there exists a run $\rho' \in \sigma(\mathcal{WS}_{1-i})$ such that $\rho(p_i) \approx \rho'(p_{1-i})$.

5 Formal Model of SPRESSO

We now present the formal model of SPRESSO, which closely follows the description in Section 2 and the implementation of the system. This model is the basis for our formal analysis of privacy and authentication properties presented in Sections 6 and 7.

| $s \in \mathcal{S}$ | script(s) |
|---------------------|-----------------|
| R^{att} | att_script |
| script_rp | script_rp |
| script_rp_redir | script_rp_redir |
| script_idp | script_idp |
| script_fwd | script_fwd |

Fig. 3. List of scripts in \mathcal{S} and their respective string representations.

We model SPRESSO as a web system (in the sense of Section 3.2). We call $\mathcal{SWS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ an *SPRESSO web system* if it is of the form described in what follows.

The set $\mathcal{W} = \text{Hon} \cup \text{Web} \cup \text{Net}$ consists of a finite set of web attacker processes (in Web), at most one network attacker process (in Net), a finite set FWD of forwarders, a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set IDP of web servers for the identity providers, and a finite set DNS of DNS servers, with $\text{Hon} := \text{BURP} \cup \text{IDP} \cup \text{FWD} \cup \text{DNS}$. Figure 7 shows the set of scripts \mathcal{S} and their respective string representations that are defined by the mapping script. The set E^0 contains only the trigger events as specified in Section 3.2.

We now sketch the processes and the scripts in \mathcal{W} and \mathcal{S} (see Appendix D for full details). As mentioned, our modeling closely follows the description in Section 2 and the implementation of the system:

- Browsers (in B) are defined as described in Section 3.3.
- A relying party (in RP) is a web server. RP knows four distinct paths: `/`, where it serves the index web page (script_rp), `/startLogin`, where it only accepts POST requests and mainly issues a fresh RP nonce, `/redir`, where it only accepts requests with a valid login session token and serves script_rp_redir to redirect the browser to the IdP, and `/login`, where it also only accepts POST requests with login data obtained during the login process by script_rp running in the browser. It checks this data and, if the data is considered to be valid, it issues a service token. The RP keeps a list of such tokens in its state. Intuitively, a client having such a token can use the service of the RP.
- Each IdP (in IDP) is a web server. It knows three distinct paths: `/.well-known/spresso-login`, where it serves the login dialog web page (script_idp), `/.well-known/spresso-info`, where it serves the support document containing its public key, and `/sign`, where it issues a (signed) identity assertion. Users can authenticate to the IdP with their credentials and IdP tracks the state of the users with sessions. Only authenticated users can receive IAs from the IdP.
- Forwarders (in FWD) are web servers that have only one state (i.e., they are stateless) and serve only the script script_fwd, except if they become corrupted.
- Each DNS server (in DNS) contains the assignment of domain names to IP addresses and answers DNS requests accordingly.

Besides the browser, RPs, IdPs, and FWDs can become corrupted: If they receive the message CORRUPT, they start collecting all incoming messages in their state and when triggered send out some message that is derivable from their state and collected input messages, just like an attacker process.

6 Privacy of SPRESSO

In our privacy analysis, we show that an identity provider in SPRESSO cannot learn where its users log in. We formalize this property as an indistinguishability property: an identity provider (modeled as a web attacker) cannot distinguish between a user logging in at one relying party and the same user logging in at a different relying party.

Definition of Privacy of SPRESSO. The web systems considered for the privacy of SPRESSO are the web systems \mathcal{SWS} defined in Section 5 which now contain one or more web attackers, no network attackers, one honest DNS server, one honest forwarder, one browser, and two honest relying parties r_1 and r_2 . All honest parties may not become corrupted and use the honest DNS server for address resolving. Identity providers are assumed to be dishonest, and hence, are subsumed by the web attackers (which govern all identities). The web attacker subsumes also potentially

dishonest forwarders, DNS servers, relying parties, and other servers. The honest relying parties are set up such that they already contain the public signing keys (used to verify identity assertions) for each domain registered at the DNS server, modeling that these have been cached by the relying parties, as discussed in Section 2.2.

In order to state the privacy property, we replace the (only) honest browser in the above described web systems by a slightly extended browser, which we call a *challenge browser*: This browser may not become corrupted and is parameterized by a domain r of a relying party. When it is to assemble an HTTP(S) request for the special domain CHALLENGE, then instead of putting together and sending out the request for CHALLENGE it takes the domain r . However, this is done only for the first request to CHALLENGE. Further requests to this domain are not altered (and would fail, as the domain CHALLENGE is not listed in the honest DNS server).

We denote web systems as described above by $\mathcal{SWS}^{priv}(r)$, where r is the domain of the relying party given to the challenge browser in this system.

We can now define privacy of SPRESSO. We note that it is not important which attacker process in $\mathcal{SWS}^{priv}(\cdot)$ is the distinguished one (in the sense of Section 4).

Definition 2. *We say that SPRESSO is IdP-private iff for every web system $\mathcal{SWS}^{priv}(\cdot)$ and domains r_1 and r_2 of relying parties as described above, we have that $\mathcal{SWS}^{priv}(r_1) \approx \mathcal{SWS}^{priv}(r_2)$, i.e., $\mathcal{SWS}^{priv}(r_1)$ and $\mathcal{SWS}^{priv}(r_2)$ are indistinguishable.*

Note that there are many different situations where the honest browser in $\mathcal{SWS}^{priv}(\cdot)$ could be triggered to send an HTTP(S) request to CHALLENGE. This could, for example, be triggered by the user who enters a URL in the location bar of the browser, a location header (e.g., determined by the adversary), an (attacker) script telling the browser to follow a link or create an iframe, etc.

Now, the above definition requires that in every stage of a run and no matter how and by whom the CHALLENGE request was triggered, no (malicious) IdP can tell whether CHALLENGE was replaced by r_1 or r_2 , i.e., whether this resulted in a login request for r_1 or r_2 . Recall that the CHALLENGE request is replaced by the honest browser only once. This is the only place in a run where the adversary does not know whether this is a request to r_1 or r_2 . Other requests in a run, even to both r_1 and r_2 , the adversary can determine. Still, he should not be able to figure out what happened in the CHALLENGE request. Hence, this definition captures in a strong sense the intuition that a malicious IdP should not be able to distinguish whether a user logs in/has logged in at r_1 or r_2 .

Analyzing Privacy of SPRESSO. The following theorem says that SPRESSO enjoys the described privacy definition.

Theorem 1. *SPRESSO is IdP-private.*

The full proof is provided in Appendix H. In the proof, we define an equivalence relation between configurations of $\mathcal{SWS}^{priv}(r_1)$ and $\mathcal{SWS}^{priv}(r_2)$, comprising equivalences between states and equivalences between events (in the pool of waiting events). For the states, for each (type of an) atomic DY process in the web system, we define how their states are related. For example, the state of the FWD server must be identical in both configurations. As another example, roughly speaking, the attacker's state is the same up to subterms the attacker cannot decrypt. Regarding (waiting) events, we distinguish between messages that result (directly or indirectly) from a CHALLENGE request by the browser and other messages. While the challenged messages may differ in certain ways, other messages may only differ in parts that the attacker cannot decrypt.

Given these equivalences, we then show by induction and an exhaustive case distinction that, starting from equivalent configurations, every schedule leads to equivalent configurations. (We note that in $\mathcal{SWS}^{priv}(\cdot)$ a schedule induces a single run because in $\mathcal{SWS}^{priv}(\cdot)$ we do not have non-deterministic actions that are not determined by a schedule: honest servers and scripts perform only deterministic actions.) As an example, we distinguish between the potential receivers of an event. If, e.g., FWD is a receiver of a message, given its identical state in both configurations (as per the equivalence definition) and the equivalence on the input event, we can immediately show that the equivalence holds on the output message and state. For other atomic DY processes, such as browsers and RPs, this is much harder to show. For example, for browsers, we need to distinguish between the different scripts that can potentially run in the browser (including the attacker script), the origins under which these scripts run, and the actions they can perform.

For equivalent configurations of $\mathcal{SWS}^{priv}(r_1)$ and $\mathcal{SWS}^{priv}(r_2)$, we show that the attacker's views are indistinguishable. Given that for all $\mathcal{SWS}^{priv}(r_1)$ and $\mathcal{SWS}^{priv}(r_2)$ every schedule leads to equivalent configurations, we have that SPRESSO is IdP-private.

7 Authentication of SPRESSO

We show that SPRESSO satisfies two fundamental authentication properties.

Formal Model of SPRESSO for Authentication. For the authentication analysis, we consider web systems as defined in Section 5 which now contain one network attacker, a finite set of browsers, a finite set of relying parties, a finite set of identity providers, and a finite set of forwarders. Browsers, forwarders, and relying parties can become corrupted by the network attacker. The network attacker subsumes all web attackers and also acts as a (dishonest) DNS server to all other parties. We denote a web system in this class of web systems by \mathcal{SWS}^{auth} .

Defining Authentication for SPRESSO. We state two fundamental authentication properties every SSO system should satisfy. These properties are adapted from [10].

Informally, these properties can be stated as follows: **(A)** *The attacker should not be able to use a service of an honest RP as an honest user.* In other words, the attacker should not get hold of (be able to derive from his current knowledge) a service token issued by an honest RP 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 (see Section 3.3). **(B)** *The attacker should not be able to authenticate an honest browser to an honest RP with an ID that is not owned by the browser (identity injection).* For both properties, we clearly have to require that the forwarder used by the honest RP is honest as well.

We call a web system \mathcal{SWS}^{auth} *secure w.r.t. authentication* if the above conditions are satisfied in all runs of the system. We refer the reader to Appendix E for the formal definition of (A) and (B).

Analyzing Authentication of SPRESSO. We prove the following theorem:

Theorem 2. *Let \mathcal{SWS}^{auth} be an SPRESSO web system as defined above. Then \mathcal{SWS}^{auth} is secure w.r.t. authentication.*

In other words, the authentication properties (A) and (B) are fulfilled for every SPRESSO web system.

For the proof, we first show some general properties of \mathcal{SWS}^{auth} . In particular, we show that encrypted communication over HTTPS between an honest relying party and an honest IdP cannot be altered by the (network) attacker, and, based on that, any honest relying party always retrieves the “correct” public signature verification key from honest IdPs. We then proceed to show that for a service token to be issued by an honest RP, a request of a specific form has to be received by the RP.

We then use these properties and the general web system properties shown in the full version of [12] to prove properties (A) and (B) separately. In both cases, we assume that the respective property is not satisfied and lead this to a contradiction. Again, the full proof is provided in Appendix F.

8 Further Related Work

As mentioned in the introduction, many SSO systems have been developed. However, unlike SPRESSO, none of them is privacy-respecting.

Besides the design and implementation of SPRESSO, the formal analysis of this system based on an expressive web model is an important part of our work. The formal treatment of the security of web applications is a young discipline. Of the few works in this area even less are based on a general model that incorporates essential mechanisms of the web. Early works in formal web security analysis (see, e.g., [3, 8, 16, 17, 25]) are based on very limited models developed specifically for the application under scrutiny. The first work to consider a general model of the web, written in the finite-state model checker Alloy, is the work by Akhawe et al. [2]. Inspired by this work, Bansal et al. [5, 6] built a more expressive model, called WebSpi, in ProVerif [7], a tool for symbolic cryptographic protocol analysis. These models have successfully been applied to web standards and applications. Recently, Kumar [18] presented a high-level Alloy model and applied it to SAML single sign-on. The web model presented in [10], which we further extend and refine here, is the most comprehensive web model to date (see also the discussion in [10]). In fact, this is the only model in which we can analyze SPRESSO. For example, other models do not incorporate a precise handling of windows, documents, or iframes; cross-document messaging (postMessages) are not included at all.

9 Conclusion

In this paper, we proposed the first privacy-respecting (web-based) SSO system, where the IdP cannot track at which RP a user logs in. Our system, SPRESSO, is open and decentralized. Users can log in at any RP with any email address with SPRESSO support, allowing for seamless and convenient integration into the usual login process. Being solely based on standard HTML5 and web features, SPRESSO can be used across browsers, platforms, and devices.

We formally prove that SPRESSO indeed enjoys strong authentication and privacy properties. This is important since, as discussed in the paper, numerous attacks on other SSO systems have been discovered. These attacks demonstrate that designing a secure SSO system is non-trivial and security flaws can easily go undetected when no rigorous analysis is carried out.

As mentioned in Section 8, there have been only very few analysis efforts, based on expressive models of the web infrastructure, on web applications in general and SSO systems in particular in the literature so far. Therefore, the analysis carried out in this paper is also of independent interest.

Our work is the first to analyze privacy properties based on an expressive web model, in fact the most expressive model to date. The general indistinguishability/privacy definition we propose, which is not tailored to any specific web application, will be useful beyond the analysis performed in this paper.

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] 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.
- [8] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.
- [9] V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: negative tests and non-determinism. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*, pages 321–330. ACM, 2011.
- [10] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *35th IEEE Symposium on Security and Privacy (S&P 2014)*, pages 673–688. IEEE Computer Society, 2014.
- [11] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. Technical Report arXiv:1411.7210, arXiv, 2014. <http://arxiv.org/abs/1411.7210>.

- [12] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *Computer Security - ESORICS 2015, 20th European Symposium on Research in Computer Security, Vienna, Austria, September 23-25, 2015*, Lecture Notes in Computer Science. Springer, 2015. To appear. Full version available at <http://arxiv.org/abs/1411.7210>.
- [13] B. Fitzpatrick, D. Recordon, et al. OpenID Authentication 2.0. Dec. 5, 2007. http://openid.net/specs/openid-authentication-2_0.html.
- [14] D. Hardt. RFC6749 - The OAuth 2.0 Authorization Framework. Oct. 2012. <http://tools.ietf.org/html/rfc6749>.
- [15] HTML5, W3C Recommendation. Oct. 28, 2014.
- [16] 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.
- [17] 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.
- [18] A. Kumar. A Lightweight Formal Approach for Analyzing Security of Web Protocols. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 192–211. Springer, 2014.
- [19] Mozilla Identity Team. Persona. <https://login.persona.org>.
- [20] T. Nitot. Persona: more privacy, better security while making developers and users happy! Beyond the Code Blog. Apr. 9, 2013. <https://blog.mozilla.org/beyond-the-code/2013/04/09/persona-beta2/>.
- [21] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In T. Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 397–412. USENIX Association, 2012.
- [22] P. Sovis, F. Kohlar, and J. Schwenk. Security Analysis of OpenID. In *Sicherheit*, volume 170 of *LNI*, pages 329–340. GI, 2010.
- [23] SPRESSO Demo Site and Source Code, 2015. <https://spresso.me>.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 399–314. USENIX Association, 2013.
- [28] Y. Zhou and D. Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 495–510. USENIX Association, 2014.

A The Web Model

In this section, we present the model of the web infrastructure as proposed in [10] and [11], along with the following changes and additions:

- The set of waiting events is replaced by a (infinite) sequence of waiting events. The sequence initially only contains an infinite number of trigger events (interleaved by receiver). All new events output by processes are added in the front of the sequence.

- We write events as terms, i.e., $(a:f:m)$ becomes $\langle a, f, m \rangle$.
- E_0 and E in runs/processing steps are now infinite sequences instead of multi-sets.
- In runs, the index of states and events (and nonces) are now written superscript (instead of subscript).
- For atomic DY processes, we replace the set of output messages by a sequence term (as defined in the equational theory) of the form $\langle \langle a, f, m \rangle, \langle a', f', m' \rangle, \dots \rangle$. Each time such a sequence is output by any DY process, its elements are prepended to the sequence of waiting events.
- We introduce a global sequence of nonces (n_1, n_2, \dots) . Whenever any DY process outputs special placeholders ν_1, ν_2, \dots (in its state or output messages), these placeholders are replaced by freshly chosen nonces from the global set of nonces.
- A similar approach applies to scripts (running inside browsers). Instead of receiving and using a fresh set of nonces each time they are called by the browser, scripts now get no dedicated set of nonces as inputs, but instead may output operators μ_1, μ_2, \dots . After the script run has finished, these are replaced by “fresh” ν placeholders by the browser (i.e., ν placeholders the browser itself does not use otherwise.)
- We therefore remove the sets of nonces from DY processes.
- We remove the function symbol $\text{extractmsg}(\cdot)$ which extracted the signed term from a signature. Instead, we added a new function symbol $\text{checksig}(\cdot, \cdot, \cdot)$ that checks that a given term was signed.
- For an accurate privacy analysis, we introduce the *Referer*¹³ header and associated document property. We also introduce the *location* document property.
- For the script command for following a link (HREF) we add the option to avoid sending the *referer* header (as a model for the `rel="noreferrer"` attribute for links in HTML5).¹⁴
- DNS responses now not only contain the IP address of the domain for which the DNS request was sent, but also the domain itself. This is a more realistic model.

A.1 Communication Model

We here present details and definitions on the basic concepts of the communication model.

Terms, Messages and Events The signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

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

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

The equational theory associated with the signature Σ is given in Figure 4.

Definition 3 (Nonces and Terms). By $X = \{x_0, x_1, \dots\}$ we denote a set of variables and by \mathcal{N} we denote an infinite set of constants (nonces) such that Σ , 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 Σ for some $n \geq 0$ and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.

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

¹³A spelling error in the early HTTP standards.

¹⁴Note that in practice, all major browsers except for the Internet Explorer support this property.

$$\begin{aligned} \text{dec}_a(\text{enc}_a(x, \text{pub}(y)), y) &= x & (1) \\ \text{dec}_s(\text{enc}_s(x, y), y) &= x & (2) \\ \text{checksig}(\text{sig}(x, y), x, \text{pub}(y)) &= \top & (3) \\ \pi_i(\langle x_1, \dots, x_n \rangle) &= x_i \text{ if } 1 \leq i \leq n & (4) \\ \pi_j(\langle x_1, \dots, x_n \rangle) &= \diamond \text{ if } j \notin \{1, \dots, n\} & (5) \end{aligned}$$

Fig. 4. Equational theory for Σ .

Definition 4 (Ground Terms, Messages, Placeholders, Protomessages). 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 $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \dots\}$ of variables (called placeholders). The set $\mathcal{M}^\nu := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of protomessages, i.e., messages that can contain placeholders.

Example 1. 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 $\text{pub}(k)$.

Definition 5 (Normal Form). Let t be a term. The normal form of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 4. For a term t , we denote its normal form as $t \downarrow$.

Definition 6 (Pattern Matching). Let pattern $\in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t matches pattern iff t can be acquired from pattern by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim \text{pattern}$.

For a term t' we write $t' | \text{pattern}$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match pattern.

Example 2. For example, for a pattern $p = \langle \top, * \rangle$ we have that $\langle \top, 42 \rangle \sim p$, $\langle \perp, 42 \rangle \not\sim p$, and

$$\langle \langle \perp, \top \rangle, \langle \top, 23 \rangle, \langle a, b \rangle, \langle \top, \perp \rangle \rangle | p = \langle \langle \top, 23 \rangle, \langle \top, \perp \rangle \rangle .$$

Definition 7 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$. By $\tau[t_1/x_1, \dots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i , for all $i \in \{1, \dots, n\}$.

Definition 8 (Events and Protoevents). An event (over IPs and \mathcal{M}) is a term of the form $\langle a, f, m \rangle$, for $a, f \in \text{IPs}$ 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. Events over IPs and \mathcal{M}^ν are called protoevents and are denoted \mathcal{E}^ν . By $2^{\mathcal{E}}$ (or $2^{\mathcal{E}^\nu}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle \rangle$, $\langle \langle a, f, m \rangle, \langle a', f', m' \rangle, \dots \rangle$, etc.).

Atomic Processes, Systems and Runs

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 9 (Generic Atomic Processes and Systems). A (generic) atomic process is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ where $I^p \subseteq \text{IPs}$, $Z^p \in \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^\nu} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p . For any new state s and any sequence of nonces (η_1, η_2, \dots) we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \dots] \in Z^p$. A system \mathcal{P} is a (possibly infinite) set of atomic processes.

Definition 10 (Configurations). A configuration of a system \mathcal{P} is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence¹⁵ (e_1, e_2, \dots) of events waiting to be delivered, and N is an infinite sequence of nonces (n_1, n_2, \dots) .

¹⁵Here: Not in the sense of terms as defined earlier.

Definition 11 (Concatenating sequences). For a term $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = (b_1, b_2, \dots)$, we define the concatenation as $a \cdot b := (a_1, \dots, a_i, b_1, b_2, \dots)$.

Definition 12 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y , a non-deterministically chosen occurrence of that element in X is removed.

Definition 13 (Processing Steps). A processing step of the system \mathcal{P} is of the form

$$(S, E, N) \xrightarrow[p \rightarrow E_{out}]{e_{in} \rightarrow p} (S', E', N')$$

where

1. (S, E, N) and (S', E', N') are configurations of \mathcal{P} ,
2. $e_{in} = \langle a, f, m \rangle \in E$ is an event,
3. $p \in \mathcal{P}$ is a process,
4. E_{out} is a sequence (term) of events

such that there exists

1. a sequence (term) $E_{out}^\nu \subseteq 2^{E^\nu}$ of protoevents,
2. a term $s^\nu \in \mathcal{T}_{\mathcal{N}}(V_{process})$,
3. a sequence (v_1, v_2, \dots, v_i) of all placeholders appearing in E_{out}^ν (ordered lexicographically),
4. a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ of the first i elements in N

with

1. $((e_{in}, S(p)), (E_{out}^\nu, s^\nu)) \in R^p$ and $a \in I^p$,
2. $E_{out} = E_{out}^\nu[m_1/v_1, \dots, m_i/v_i]$
3. $S'(p) = s^\nu[m_1/v_1, \dots, m_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$
4. $E' = E_{out} \cdot (E \setminus \{e_{in}\})$
5. $N' = N \setminus N^\nu$

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

Intuitively, for a processing step, we select one of the processes in \mathcal{P} , and call it with one of the events in the list of waiting events E . In its output (new state and output events), we replace any occurrences of placeholders v_x by “fresh” nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

Definition 14 (Runs). Let \mathcal{P} be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A run ρ of a system \mathcal{P} initiated by E^0 with nonces N^0 is a finite sequence of configurations $((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \dots)$ such that $S^0(p) = s_0^p$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 that contains infinite trigger events of the form $\langle a, a, \text{TRIGGER} \rangle$ for each $a \in I^p$ s, interleaved by address.

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.

Definition 15 (Deriving Terms). Let M be a set of ground terms. We say that a term m can be derived from M with placeholders V if there exist $n \geq 0$, $m_1, \dots, m_n \in M$, and $\tau \in \mathcal{T}_0(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V .

For example, $a \in d_{\{\}}(\{\text{enc}_a(\langle a, b, c \rangle), \text{pub}(k), k\})$.

Definition 16 (Atomic Dolev-Yao Process). An atomic Dolev-Yao process (or simply, a DY process) is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that (I^p, Z^p, R^p, s_0^p) is an atomic process and (1) $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ (and hence, $s_0^p \in \mathcal{T}_{\mathcal{N}}$), and (2) for all events $e \in \mathcal{E}$, sequences of protoevents $E, s \in \mathcal{T}_{\mathcal{N}}, s' \in \mathcal{T}_{\mathcal{N}}(V_{process})$, with $((e, s), (E, s')) \in R^p$ it holds true that $E, s' \in d_{V_{process}}(\{e, s\})$.

Definition 17 (Atomic Attacker Process). An (atomic) attacker process for a set of sender addresses $A \subseteq \text{IPs}$ is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e , and $s \in \mathcal{T}_{\mathcal{N}}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s \rangle$ and $E = \langle \langle a_1, f_1, m_1 \rangle, \dots, \langle a_n, f_n, m_n \rangle \rangle$ with $n \in \mathbb{N}$, $a_1, \dots, a_n \in \text{IPs}$, $f_0, \dots, f_n \in A$, $m_1, \dots, m_n \in d_{V_{process}}(\{e, s\})$.

A.2 Scripting Processes

We define scripting processes, which model client-side scripting technologies, such as JavaScript. Scripting processes are defined similarly to DY processes.

Definition 18 (Placeholders for Scripting Processes). By $V_{script} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripting processes.

Definition 19 (Scripting Processes). A scripting process (or simply, a script) is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{script})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}, s' \in \mathcal{T}_{\mathcal{N}}(V_{script})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{script}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last state and limited information about the browser's state) s . The script then outputs a term s' , which represents the new internal state and some command which is interpreted by the browser. The term s' may contain variables λ_1, \dots which the browser will replace by (otherwise unused) placeholders ν_1, \dots which will be replaced by nonces once the browser DY process finishes (effectively providing the script with a way to get "fresh" nonces).

Similarly to an attacker process, we define the *attacker script* R^{att} :

Definition 20 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{script}}(s)\}$.

A.3 Web System

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Definition 21. 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.

Every $p \in \text{Web} \cup \text{Net}$ is an attacker process for some set of sender addresses $A \subseteq \text{IPs}$. 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 IPs).

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 $\text{script}(s)$.

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \text{TRIGGER} \rangle$ 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 .

B Message and Data Formats

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

B.1 Notations

Definition 22 (Sequence Notations). For a sequence $t = \langle t_1, \dots, t_n \rangle$ and a set s 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 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 is chosen arbitrarily.)

Definition 23. 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, i.e., $[] = \langle \rangle$. 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 $z[k] := \langle \rangle$.

$$[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n][k_i] = v_i \tag{6}$$

$$\begin{aligned} [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1}, \dots, k_n : v_n] - k_i = \\ [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1}, \dots, k_n : v_n] \end{aligned} \tag{7}$$

Fig. 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 π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 24. 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 τ . This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

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

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 4.

B.2 URLs

Definition 25. A URL is a term of the form $\langle URL, protocol, host, path, parameters \rangle$ with $protocol \in \{P, S\}$ (for *plain* (HTTP) and *secure* (HTTPS)), $host \in \text{Doms}$, $path \in \mathbb{S}$ and $parameters \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$. The set of all valid URLs is URLs .

Example 4. For the URL $u = \langle URL, a, b, c, d \rangle$, $u.protocol = a$. If, in the algorithm described later, we say $u.path := e$ then $u = \langle URL, a, b, c, e \rangle$ afterwards.

B.3 Origins

Definition 26. An origin is a term of the form $\langle \text{host}, \text{protocol} \rangle$ with $\text{host} \in \text{Doms}$ and $\text{protocol} \in \{P, S\}$. We write Origins for the set of all origins.

Example 5. For example, $\langle \text{F00}, S \rangle$ is the HTTPS origin for the domain F00, while $\langle \text{BAR}, P \rangle$ is the HTTP origin for the domain BAR.

B.4 Cookies

Definition 27. A cookie is a term of the form $\langle \text{name}, \text{content} \rangle$ where $\text{name} \in \mathcal{T}_{\mathcal{N}}$, and 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 Cookies for the set of all cookies and Cookies^ν for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections. If the *session* flag is set, this cookie will be deleted as soon as the browser is closed. The *httpOnly* attribute controls whether JavaScript has access to this cookie.

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

B.5 HTTP Messages

Definition 28. An HTTP request is a term of the form shown in (8). An HTTP response is a term of the form shown in (9).

$$\langle \text{HTTPReq}, \text{nonce}, \text{method}, \text{host}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \quad (8)$$

$$\langle \text{HTTPResp}, \text{nonce}, \text{status}, \text{headers}, \text{body} \rangle \quad (9)$$

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.
- $\text{host} \in \text{Doms}$ 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 $l \in \text{URLs}$
 - $\langle \text{Referer}, r \rangle$ where $r \in \text{URLs}$
 - $\langle \text{Strict-Transport-Security}, \top \rangle$
- $\text{body} \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

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

Example 6 (HTTP Request and Response).

$$r := \langle \text{HTTPReq}, n_1, \text{POST}, \text{example.com}, / \text{show}, \langle \langle \text{index}, 1 \rangle \rangle, \text{Origin} : \langle \text{example.com}, S \rangle, \langle \text{foo}, \text{bar} \rangle \rangle \quad (10)$$

$$s := \langle \text{HTTPResp}, n_1, 200, \langle \langle \text{Set-Cookie}, \langle \langle \text{SID}, \langle n_2, \perp, \perp, \top \rangle \rangle \rangle \rangle, \langle \text{somescript}, x \rangle \rangle \quad (11)$$

An HTTP GET request for the URL <http://example.com/show?index=1> is shown in (10), with an Origin header and a body that contains $\langle \text{foo}, \text{bar} \rangle$. A possible response is shown in (11), 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 supposed to encrypt the response using the symmetric key.

Definition 29. An encrypted HTTP request is of the form $\text{enc}_a(\langle m, k' \rangle, k)$, where $k, k' \in \mathcal{N}$ 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 HTTPSRequests or HTTPSResponses , respectively.

Example 7.

$$\text{enc}_a(\langle r, k' \rangle, \text{pub}(k_{\text{example.com}})) \quad (12)$$

$$\text{enc}_s(s, k') \quad (13)$$

The term (12) shows an encrypted request (with r as in (10)). It is encrypted using the public key $\text{pub}(k_{\text{example.com}})$. The term (13) is a response (with s as in (11)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (12).

B.6 DNS Messages

Definition 30. A DNS request is a term of the form $\langle \text{DNSResolve}, \text{domain}, n \rangle$ where $\text{domain} \in \text{Doms}$, $n \in \mathcal{N}$. We call the set of all DNS requests DNSRequests .

Definition 31. A DNS response is a term of the form $\langle \text{DNSResolved}, \text{domain}, \text{result}, n \rangle$ with $\text{domain} \in \text{Doms}$, $\text{result} \in \text{IPs}$, $n \in \mathcal{N}$. We call the set of all DNS responses DNSResponses .

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

B.7 DNS Servers

Here, 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.

Definition 32. A DNS server d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. 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 (otherwise ignored).

C Detailed Description of the Browser Model

Following the informal description of the browser model in Section 3.3, we now present a formal model. We start by introducing some notation and terminology.

C.1 Notation and Terminology (Web Browser State)

Before we can define the state of a web browser, we first have to define windows and documents.

Definition 33. A window is a term of the form $w = \langle \text{nonce}, \text{documents}, \text{opener} \rangle$ with $\text{nonce} \in \mathcal{N}$, $\text{documents} \subset \langle \rangle \text{Documents}$ (defined below), $\text{opener} \in \mathcal{N} \cup \{\perp\}$ where $d.\text{active} = \top$ for exactly one $d \in \langle \rangle \text{documents}$ if documents is not empty (we then call d the active document of w). We write 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*.

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.

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., $b.opener = a.nonce$.

Definition 34. A document d is a term of the form

$$\langle nonce, location, referrer, script, scriptstate, scriptinputs, subwindows, active \rangle$$

where $nonce \in \mathcal{N}$, $location \in \text{URLs}$, $referrer \in \text{URLs} \cup \{\perp\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\diamond} \text{Windows}$, $active \in \{\top, \perp\}$. A limited document is a term of the form $\langle nonce, subwindows \rangle$ with $nonce, subwindows$ as above. A window $w \in^{\diamond} subwindows$ is called a subwindow (of d). We write Documents for the set of all documents. For a document term d we write $d.origin$ to denote the origin of the document, i.e., the term $\langle d.location.host, d.location.protocol \rangle \in \text{Origins}$.

We will refer to the document nonce as (*document*) *reference*.

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 23.

Definition 35. The set of states Z^p of a web browser atomic process p consists of the terms of the form

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, sts, DNSaddress, pendingDNS, pendingRequests, isCorrupted \rangle$$

where

- $windows \subset^{\diamond} \text{Windows}$,
- $ids \subset^{\diamond} \mathcal{T}_{\mathcal{N}}$,
- $secrets \in [\text{Origins} \times \mathcal{N}]$,
- $cookies$ is a dictionary over Doms and dictionaries of Cookies,
- $localStorage \in [\text{Origins} \times \mathcal{T}_{\mathcal{N}}]$,
- $sessionStorage \in [OR \times \mathcal{T}_{\mathcal{N}}]$ for $OR := \{\langle o, r \rangle \mid o \in \text{Origins}, r \in \mathcal{N}\}$,
- $keyMapping \in [\text{Doms} \times \mathcal{T}_{\mathcal{N}}]$,
- $sts \subset^{\diamond} \text{Doms}$,
- $DNSaddress \in \text{IPs}$,
- $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$,
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$,
- and $isCorrupted \in \{\perp, \text{FULLCORRUPT}, \text{CLOSECORRUPT}\}$.

Definition 36. For two window terms w and w' we write $w \xrightarrow{\text{childof}} w'$ if

$$w \in^{\diamond} w'.activedocument.subwindows.$$

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 , $\text{Subwindows}(s)$ denotes the set of all pointers¹⁶ to windows in the window list $s.windows$, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With $\text{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\text{Subwindows}(s)$.

¹⁶Recall the definition of a pointer in Definition 24.

Definition 37. For a browser state s we denote by $\text{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in \langle \rangle s.\text{windows}$ 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 $\text{Docs}(s)$ 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 $s.\bar{p}' = s.\bar{p}.\text{activedocument}$.

By $\text{Subwindows}^+(s)$ and $\text{Docs}^+(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

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 38. Let s be a browser state and d a document. By $\text{Clean}(s, d)$ 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 output 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 39. For a sequence of cookies (with pairwise different names) *oldcookies* and a sequence of cookies *newcookies*, the set $\text{CookieMerge}(\text{oldcookies}, \text{newcookies})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\text{content.httpOnly} \equiv \top$. For any $c, c' \in \langle \rangle \text{newcookies}$, $c.\text{name} \equiv c'.\text{name}$, 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 $c_{\text{old}} \in \langle \rangle \text{oldcookies}$, $c_{\text{new}} \in \langle \rangle \text{newcookies}$, $c_{\text{old}}.\text{name} \equiv c_{\text{new}}.\text{name}$, add c_{new} to m if $c_{\text{old}}.\text{content.httpOnly} \equiv \perp$ and add c_{old} to m otherwise. The result of $\text{CookieMerge}(\text{oldcookies}, \text{newcookies})$ is m .

The function `AddCookie` adds a cookie c received in an HTTP response 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 40. For a sequence of cookies (with pairwise different names) *oldcookies* and a cookie c , the sequence $\text{AddCookie}(\text{oldcookies}, c)$ 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 .

The function `NavigableWindows` returns a set of windows that a document is allowed to navigate. We closely follow [15], Section 5.1.4 for this definition.

Definition 41. The set $\text{NavigableWindows}(\bar{w}, s')$ 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 $s'.\bar{w}'.\text{activedocument.origin} \equiv s'.\bar{w}.\text{activedocument.origin}$ (i.e., 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 $\bar{w}' \in \bar{W}$.

C.2 Description of the Web Browser Atomic Process

We will now describe the relation R^p of a standard HTTP browser p . We define $(\langle\langle\langle a, f, m \rangle\rangle, s \rangle, (M, s'))$ to belong to R^p iff the non-deterministic algorithm presented below, when given $(\langle a, f, m \rangle, s)$ as input, terminates with **stop** M, s' , i.e., with output M and s' . Recall that $\langle a, f, m \rangle$ is an (input) event and s is a (browser) state, M is a sequence of (output) protoevents, and s' is a new (browser) state (potentially with placeholders for nonces).

Notations. 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 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.

Placeholders. In several places throughout the algorithms presented next we use placeholders to generate “fresh” nonces as described in our communication model (see Definition 3). Figure 6 shows a list of all placeholders used.

| Placeholder | Usage |
|-------------------|---|
| ν_1 | Algorithm 7, new window nonces |
| ν_2 | Algorithm 7, new HTTP request nonce |
| ν_3 | Algorithm 7, lookup key for pending HTTP requests entry |
| ν_4 | Algorithm 5, new HTTP request nonce (multiple lines) |
| ν_5 | Algorithm 5, new subwindow nonce |
| ν_6 | Algorithm 6, new HTTP request nonce |
| ν_7 | Algorithm 6, new document nonce |
| ν_8 | Algorithm 4, lookup key for pending DNS entry |
| ν_9 | Algorithm 1, new window nonce |
| ν_{10}, \dots | Algorithm 5, replacement for placeholders in scripting process output |

Fig. 6. List of placeholders used in browser algorithms.

Before we describe the main browser algorithm, we first define some functions.

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

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 1 Determine window for navigation.

```

1: function GETNAVIGABLEWINDOW( $\bar{w}, window, noreferrer, s'$ )
2:   if  $window \equiv \_BLANK$  then                                     ▷ Open a new window when _BLANK is used
3:     if  $noreferrer \equiv \perp$  then
4:       let  $w' := \langle \nu_9, \langle \rangle, s'.\bar{w}.nonce \rangle$ 
5:       else

```



```

6:     let  $w' := \langle \nu_9, \langle \rangle, \perp \rangle$ 
7:     end if
8:     let  $s'.windows := s'.windows +^{\langle \rangle} w'$  and let  $\bar{w}'$  be a pointer to this new element in  $s'$ 
9:     return  $\bar{w}'$ 
10:  end if
11:  let  $\bar{w}' \leftarrow \text{NavigableWindows}(\bar{w}, s')$  such that  $s'.\bar{w}'.nonce \equiv window$  if possible; otherwise return  $\bar{w}$ 
12:  return  $\bar{w}'$ 
13: 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 2 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  $\bar{w}$ 
3:   if  $s'.\bar{w}'.activedocument.origin \equiv s'.\bar{w}.activedocument.origin$  then
4:     return  $\bar{w}'$ 
5:   end if
6:   return  $\bar{w}$ 
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 3 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 4 Prepare headers, do DNS resolution, save message.

```

1: function SEND( $reference, message, protocol, origin, referrer, s'$ )
2:   if  $message.host \in^{\langle \rangle} s'.sts$  then
3:     let  $protocol := S$ 
4:   end if
5:   let  $cookies := \{ \langle c.name, c.content.value \rangle \mid c \in^{\langle \rangle} s'.cookies[message.host] \}$ 
    $\hookrightarrow \wedge (c.content.secure \implies (protocol = S))$ 
6:   let  $message.headers[Cookie] := cookies$ 
7:   if  $origin \neq \perp$  then
8:     let  $message.headers[Origin] := origin$ 
9:   end if
10:  if  $referrer \neq \perp$  then
11:    let  $message.headers[Referer] := referrer$ 
12:  end if
13:  let  $s'.pendingDNS[\nu_8] := \langle reference, message, protocol \rangle$ 
14:  stop  $\langle \langle s'.DNSaddress, a, \langle DNSResolve, host, n \rangle \rangle, s' \rangle$ 
15: end function

```


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.

Algorithm 5 Execute a script.

```

1: function RUNSCRIPT( $\bar{w}, \bar{d}, s'$ )
2:   let  $tree := \text{Clean}(s', s'.\bar{d})$ 
3:   let  $cookies := \langle \{ \langle c.name, c.content.value \rangle \mid c \in {}^\diamond s'.cookies [s'.\bar{d}.origin.host] \}$ 
       $\hookrightarrow \wedge c.content.httpOnly = \perp$ 
       $\hookrightarrow \wedge (c.content.secure \implies (s'.\bar{d}.origin.protocol \equiv S)) \rangle$ 
4:   let  $tlw \leftarrow s'.windows$  such that  $tlw$  is the top-level window containing  $\bar{d}$ 
5:   let  $sessionStorage := s'.sessionStorage [\langle s'.\bar{d}.origin, tlw.nonce \rangle]$ 
6:   let  $localStorage := s'.localStorage [s'.\bar{d}.origin]$ 
7:   let  $secret := s'.secrets [s'.\bar{d}.origin]$ 
8:   let  $R \leftarrow \text{script}^{-1}(s'.\bar{d}.script)$ 
9:   let  $in := \langle tree, s'.\bar{d}.nonce, s'.\bar{d}.scriptstate, s'.\bar{d}.scriptinputs, cookies, localStorage, sessionStorage, s'.ids, secret \rangle$ 
10:  let  $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$ ,
       $\hookrightarrow cookies' \leftarrow \text{Cookies}'$ ,
       $\hookrightarrow localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$ ,
       $\hookrightarrow sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$ ,
       $\hookrightarrow command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$ ,
       $\hookrightarrow out^\lambda := \langle state', cookies', localStorage', sessionStorage', command \rangle$ 
       $\hookrightarrow$  such that  $(in, out^\lambda) \in R$ 
11:  let  $out := out^\lambda [\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$ 
12:  let  $s'.cookies [s'.\bar{d}.origin.host] := \langle \text{CookieMerge}(s'.cookies [s'.\bar{d}.origin.host], cookies') \rangle$ 
13:  let  $s'.localStorage [s'.\bar{d}.origin] := localStorage'$ 
14:  let  $s'.sessionStorage [\langle s'.\bar{d}.origin, tlw.nonce \rangle] := sessionStorage'$ 
15:  let  $s'.\bar{d}.scriptstate := state'$ 
16:  switch  $command$  do
17:    case  $\langle \text{HREF}, url, hrefwindow, noreferrer \rangle$ 
18:      let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, noreferrer, s')$ 
19:      let  $req := \langle \text{HTTPReq}, \nu_4, \text{GET}, url.host, url.path, \langle \rangle, url.parameters, \langle \rangle \rangle$ 
20:      if  $noreferrer \equiv \perp$  then
21:        let  $referrer := s'.\bar{d}.location$ 
22:      else
23:        let  $referrer := \perp$ 
24:      end if
25:      let  $s' := \text{CANCELNAV}(s'.\bar{w}'.nonce, s')$ 
26:       $\text{SEND}(s'.\bar{w}'.nonce, req, url.protocol, \perp, referrer, s')$ 
27:    case  $\langle \text{IFRAME}, url, window \rangle$ 
28:      let  $\bar{w}' := \text{GETWINDOW}(\bar{w}, window, s')$ 
29:      let  $req := \langle \text{HTTPReq}, \nu_4, \text{GET}, url.host, url.path, \langle \rangle, url.parameters, \langle \rangle \rangle$ 
30:      let  $referrer := s'.\bar{w}'.activedocument.location$ 
31:      let  $w' := \langle \nu_5, \langle \rangle, \perp \rangle$ 
32:      let  $s'.\bar{w}'.activedocument.subwindows := s'.\bar{w}'.activedocument.subwindows + {}^\diamond w'$ 
33:       $\text{SEND}(\nu_5, req, url.protocol, \perp, referrer, s')$ 
34:    case  $\langle \text{FORM}, url, method, data, hrefwindow \rangle$ 
35:      if  $method \notin \{ \text{GET}, \text{POST} \}$  then 17
36:        stop  $\langle \rangle, s'$ 
37:      end if
38:      let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, hrefwindow, \perp, s')$ 

```

¹⁷The 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>.

```

39:   if method = GET then
40:     let body :=  $\langle \rangle$ 
41:     let parameters := data
42:     let origin :=  $\perp$ 
43:   else
44:     let body := data
45:     let parameters := url.parameters
46:     let origin :=  $s'.\bar{d}.\text{origin}$ 
47:   end if
48:   let req :=  $\langle \text{HTTPReq}, \nu_4, \text{method}, \text{url.host}, \text{url.path}, \langle \rangle, \text{parameters}, \text{body} \rangle$ 
49:   let referrer :=  $s'.\bar{d}.\text{location}$ 
50:   let  $s'$  := CANCELNAV( $s'.\bar{w}'.\text{nonce}, s'$ )
51:   SEND( $s'.\bar{w}'.\text{nonce}, \text{req}, \text{url.protocol}, \text{origin}, \text{referrer}, s'$ )
52: case  $\langle \text{SETSCRIPT}, \text{window}, \text{script} \rangle$ 
53:   let  $\bar{w}'$  := GETWINDOW( $\bar{w}, \text{window}, s'$ )
54:   let  $s'.\bar{w}'.\text{activedocument.script}$  := script
55:   stop  $\langle \rangle, s'$ 
56: case  $\langle \text{SETSCRIPTSTATE}, \text{window}, \text{scriptstate} \rangle$ 
57:   let  $\bar{w}'$  := GETWINDOW( $\bar{w}, \text{window}, s'$ )
58:   let  $s'.\bar{w}'.\text{activedocument.scriptstate}$  := scriptstate
59:   stop  $\langle \rangle, s'$ 
60: case  $\langle \text{XMLHTTPREQUEST}, \text{url}, \text{method}, \text{data}, \text{xhrreference} \rangle$ 
61:   if method  $\in \{ \text{CONNECT}, \text{TRACE}, \text{TRACK} \} \wedge \text{xhrreference} \notin \{ \mathcal{N}, \perp \}$  then
62:     stop  $\langle \rangle, s'$ 
63:   end if
64:   if  $\text{url.host} \neq s'.\bar{d}.\text{origin.host} \vee \text{url.protocol} \neq s'.\bar{d}.\text{origin.protocol}$  then
65:     stop  $\langle \rangle, s'$ 
66:   end if
67:   if method  $\in \{ \text{GET}, \text{HEAD} \}$  then
68:     let data :=  $\langle \rangle$ 
69:     let origin :=  $\perp$ 
70:   else
71:     let origin :=  $s'.\bar{d}.\text{origin}$ 
72:   end if
73:   let req :=  $\langle \text{HTTPReq}, \nu_4, \text{method}, \text{url.host}, \text{url.path}, \text{url.parameters}, \text{data} \rangle$ 
74:   let referrer :=  $s'.\bar{d}.\text{location}$ 
75:   SEND( $\langle s'.\bar{d}.\text{nonce}, \text{xhrreference} \rangle, \text{req}, \text{url.protocol}, \text{origin}, \text{referrer}, s'$ )
76: case  $\langle \text{BACK}, \text{window} \rangle$ 18
77:   let  $\bar{w}'$  := GETNAVIGABLEWINDOW( $\bar{w}, \text{window}, \perp, s'$ )
78:   if  $\exists \bar{j} \in \mathbb{N}, \bar{j} > 1$  such that  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active} \equiv \top$  then
79:     let  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active}$  :=  $\perp$ 
80:     let  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active}$  :=  $\top$ 
81:     let  $s'$  := CANCELNAV( $s'.\bar{w}'.\text{nonce}, s'$ )
82:   end if
83:   stop  $\langle \rangle, s'$ 
84: case  $\langle \text{FORWARD}, \text{window} \rangle$ 
85:   let  $\bar{w}'$  := GETNAVIGABLEWINDOW( $\bar{w}, \text{window}, \perp, s'$ )
86:   if  $\exists \bar{j} \in \mathbb{N}$  such that  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active} \equiv \top \wedge s'.\bar{w}'.\text{documents}.\bar{j}.\text{active} \in \text{Documents}$  then
87:     let  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active}$  :=  $\perp$ 
88:     let  $s'.\bar{w}'.\text{documents}.\bar{j}.\text{active}$  :=  $\top$ 
89:     let  $s'$  := CANCELNAV( $s'.\bar{w}'.\text{nonce}, s'$ )

```

¹⁸Note 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 reload can be triggered non-deterministically at any point (in the main algorithm).

```

90:     end if
91:     stop  $\langle \rangle, s'$ 
92:   case  $\langle \text{CLOSE}, window \rangle$ 
93:     let  $\bar{w}' := \text{GETNAVIGABLEWINDOW}(\bar{w}, window, \perp, s')$ 
94:     remove  $s'.\bar{w}'$  from the sequence containing it
95:     stop  $\langle \rangle, s'$ 
96:   case  $\langle \text{POSTMESSAGE}, window, message, origin \rangle$ 
97:     let  $\bar{w}' \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}'.nonce \equiv window$ 
98:     if  $\exists \bar{j} \in \mathbb{N}$  such that  $s'.\bar{w}'.documents.\bar{j}.active \equiv \top$ 
99:        $\hookrightarrow \wedge (origin \neq \perp \implies s'.\bar{w}'.documents.\bar{j}.origin \equiv origin)$  then
100:         let  $s'.\bar{w}'.documents.\bar{j}.scriptinputs$ 
101:            $\hookrightarrow := s'.\bar{w}'.documents.\bar{j}.scriptinputs$ 
102:            $\hookrightarrow + \langle \text{POSTMESSAGE}, s'.\bar{w}.nonce, s'.\bar{d}.origin, message \rangle$ 
103:         end if
104:       stop  $\langle \rangle, s'$ 
105:     case else
106:       stop  $\langle \rangle, s'$ 
107:   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 a window or a document reference is given (see explanation for Algorithm 4 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 6 Process an HTTP response.

```

1: function PROCESSRESPONSE(response, reference, request, protocol, s')
2:   if Set-Cookie  $\in$  response.headers then
3:     for each  $c \in \langle \rangle$  response.headers[Set-Cookie],  $c \in$  Cookies do
4:       let  $s'.cookies[request.url.host] := \text{AddCookie}(s'.cookies[request.url.host], c)$ 
5:     end for
6:   end if
7:   if Strict-Transport-Security  $\in$  response.headers  $\wedge$  protocol  $\equiv$  S then
8:     let  $s'.sts := s'.sts + \langle \rangle request.host$ 
9:   end if
10:  if Referer  $\in$  request.headers then
11:    let referrer := request.headers[Referer]
12:  else
13:    let referrer :=  $\perp$ 
14:  end if
15:  if Location  $\in$  response.headers  $\wedge$  response.status  $\in$  {303, 307} then19
16:    let url := response.headers[Location]
17:    let method' := request.method20
18:    let body' := request.body21

```

¹⁹The 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.

²⁰While the standard demands that users confirm redirections of non-safe-methods (e.g., POST), we assume that users generally confirm these redirections.

²¹If, 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.

```

19:   if  $\text{Origin} \in \text{request.headers}$  then
20:     let  $\text{origin} := \langle \text{request.headers}[\text{Origin}], \langle \text{request.host}, \text{protocol} \rangle \rangle$ 
21:   else
22:     let  $\text{origin} := \perp$ 
23:   end if
24:   if  $\text{response.status} \equiv 303 \wedge \text{request.method} \notin \{\text{GET}, \text{HEAD}\}$  then
25:     let  $\text{method}' := \text{GET}$ 
26:     let  $\text{body}' := \langle \rangle$ 
27:   end if
28:   if  $\nexists \bar{w} \in \text{Subwindows}(s')$  such that  $s'.\bar{w}.\text{nonce} \equiv \text{reference}$  then ▷ Do not redirect XHRs.
29:     stop  $\langle \rangle, s$ 
30:   end if
31:   let  $\text{req} := \langle \text{HTTPReq}, \nu_6, \text{method}', \text{url.host}, \text{url.path}, \langle \rangle, \text{url.parameters}, \text{body}' \rangle$ 
32:    $\text{SEND}(\text{reference}, \text{req}, \text{url.protocol}, \text{origin}, \text{referrer}, s')$ 
33: end if
34: if  $\exists \bar{w} \in \text{Subwindows}(s')$  such that  $s'.\bar{w}.\text{nonce} \equiv \text{reference}$  then ▷ normal response
35:   let  $\text{location} := \langle \text{URL}, \text{protocol}, \text{request.host}, \text{request.path}, \text{request.parameters} \rangle$ 
36:   if  $\text{response.body} \not\sim \langle *, * \rangle$  then
37:     stop  $\{ \}, s'$ 
38:   end if
39:   let  $\text{script} := \pi_1(\text{response.body})$ 
40:   let  $\text{scriptstate} := \pi_2(\text{response.body})$ 
41:   let  $d := \langle \nu_7, \text{location}, \text{referrer}, \text{script}, \text{scriptstate}, \langle \rangle, \langle \rangle, \top \rangle$ 
42:   if  $s'.\bar{w}.\text{documents} \equiv \langle \rangle$  then
43:     let  $s'.\bar{w}.\text{documents} := \langle d \rangle$ 
44:   else
45:     let  $\bar{i} \leftarrow \mathbb{N}$  such that  $s'.\bar{w}.\text{documents}.\bar{i}.\text{active} \equiv \top$ 
46:     let  $s'.\bar{w}.\text{documents}.\bar{i}.\text{active} := \perp$ 
47:     remove  $s'.\bar{w}.\text{documents}.\bar{i} + 1$  and all following documents from  $s'.\bar{w}.\text{documents}$ 
48:     let  $s'.\bar{w}.\text{documents} := s'.\bar{w}.\text{documents} + \langle \rangle d$ 
49:   end if
50:   stop  $\{ \}, s'$ 
51: else if  $\exists \bar{w} \in \text{Subwindows}(s'), \bar{d}$  such that  $s'.\bar{d}.\text{nonce} \equiv \pi_1(\text{reference})$ 
▷ process XHR response
↔  $\wedge s'.\bar{d} = s'.\bar{w}.\text{activedocument}$  then
52:   let  $s'.\bar{d}.\text{scriptinputs} := s'.\bar{d}.\text{scriptinputs} + \langle \rangle \langle \text{XMLHTTPREQUEST}, \text{response.body}, \pi_2(\text{reference}) \rangle$ 
53: end if
54: end function

```

Main Algorithm. This is the main algorithm of the browser relation. It receives the message m as input, as well as a , f and s as above.

Algorithm 7 Main Algorithm

```

Input:  $\langle a, f, m \rangle, s$ 
1: let  $s' := s$ 
2: if  $s.\text{isCorrupted} \neq \perp$  then
3:   let  $s'.\text{pendingRequests} := \langle m, s.\text{pendingRequests} \rangle$  ▷ Collect incoming messages
4:   let  $m' \leftarrow d_V(s')$ 
5:   let  $a' \leftarrow \text{IPs}$ 
6:   stop  $\langle \langle a', a, m' \rangle \rangle, s'$ 
7: end if
8: if  $m \equiv \text{TRIGGER}$  then ▷ A special trigger message.
9:   let  $\text{switch} \leftarrow \{1, 2, 3\}$ 
10:  if  $\text{switch} \equiv 1$  then ▷ Run some script.
11:    let  $\bar{w} \leftarrow \text{Subwindows}(s')$  such that  $s'.\bar{w}.\text{documents} \neq \langle \rangle$  if possible; otherwise stop  $\langle \rangle, s'$ 
12:    let  $\bar{d} := \bar{w} + \langle \rangle \text{activedocument}$ 
13:     $\text{RUNSCRIPT}(\bar{w}, \bar{d}, s')$ 

```

```

14: else if  $switch \equiv 2$  then ▷ Create some new request.
15:   let  $w' := \langle \nu_1, \langle \rangle, \perp \rangle$ 
16:   let  $s'.windows := s'.windows + \langle \rangle w'$ 
17:   let  $protocol \leftarrow \{P, S\}$ 
18:   let  $host \leftarrow Doms$ 
19:   let  $path \leftarrow S$ 
20:   let  $parameters \leftarrow [S \times S]$ 
21:   let  $req := \langle HTTPReq, \nu_2, GET, host, path, \langle \rangle, parameters, \langle \rangle \rangle$ 
22:   SEND( $\nu_1, req, protocol, \perp, s'$ )
23: else if  $switch \equiv 3$  then ▷ Reload some document.
24:   let  $\bar{w} \leftarrow Subwindows(s')$  such that  $s'.\bar{w}.documents \neq \langle \rangle$  if possible; otherwise stop  $\langle \rangle, s'$ 
25:   let  $url := s'.\bar{w}.activedocument.location$ 
26:   let  $req := \langle HTTPReq, \nu_2, GET, url.host, url.path, \langle \rangle, url.parameters, \langle \rangle \rangle$ 
27:   let  $referrer := s'.\bar{w}.activedocument.referrer$ 
28:   let  $s' := CANCELNAV(s'.\bar{w}.nonce, s')$ 
29:   SEND( $s'.\bar{w}.nonce, req, url.protocol, \perp, referrer, s'$ )
30: end if
31: else if  $m \equiv FULLCORRUPT$  then ▷ Request to corrupt browser
32:   let  $s'.isCorrupted := FULLCORRUPT$ 
33:   stop  $\langle \rangle, s'$ 
34: else if  $m \equiv CLOSECORRUPT$  then ▷ Close the browser
35:   let  $s'.secrets := \langle \rangle$ 
36:   let  $s'.windows := \langle \rangle$ 
37:   let  $s'.pendingDNS := \langle \rangle$ 
38:   let  $s'.pendingRequests := \langle \rangle$ 
39:   let  $s'.sessionStorage := \langle \rangle$ 
40:   let  $s'.cookies \subset \langle \rangle Cookies$  such that  $(c \in \langle \rangle s'.cookies) \iff (c \in \langle \rangle s.cookies \wedge c.content.session \equiv \perp)$ 
41:   let  $s'.isCorrupted := CLOSECORRUPT$ 
42:   stop  $\langle \rangle, s'$ 
43: else if  $\exists \langle reference, request, key, f \rangle \in \langle \rangle s'.pendingRequests$ 
    $\hookrightarrow$  such that  $\pi_1(dec_s(m, key)) \equiv HTTPResp$  then ▷ Encrypted HTTP response
44:   let  $m' := dec_s(m, key)$ 
45:   if  $m'.nonce \neq request.nonce$  then
46:     stop  $\langle \rangle, s$ 
47:   end if
48:   remove  $\langle reference, request, key, f \rangle$  from  $s'.pendingRequests$ 
49:   PROCESSRESPONSE( $m', reference, request, S, s'$ )
50: else if  $\pi_1(m) \equiv HTTPResp \wedge \exists \langle reference, request, \perp, f \rangle \in \langle \rangle s'.pendingRequests$  such that  $m'.nonce \equiv request.key$  then
51:   remove  $\langle reference, request, \perp, f \rangle$  from  $s'.pendingRequests$ 
52:   PROCESSRESPONSE( $m, reference, request, P, s'$ )
53: else if  $m \in DNSResponses$  then ▷ Successful DNS response
54:   if  $m.nonce \notin s.pendingDNS \vee m.result \notin IPs \vee m.domain \neq \pi_2(s.pendingDNS).host$  then
55:     stop  $\langle \rangle, s$ 
56:   end if
57:   let  $\langle reference, message, protocol \rangle := s.pendingDNS[m.nonce]$ 
58:   if  $protocol \equiv S$  then
59:     let  $s'.pendingRequests := s'.pendingRequests + \langle \rangle \langle reference, message, \nu_3, m.result \rangle$ 
60:     let  $message := enc_a(\langle message, \nu_3 \rangle, s'.keyMapping[message.host])$ 
61:   else
62:     let  $s'.pendingRequests := s'.pendingRequests + \langle \rangle \langle reference, message, \perp, m.result \rangle$ 
63:   end if
64:   let  $s'.pendingDNS := s'.pendingDNS - m.nonce$ 
65:   stop  $\langle \langle m.result, a, message \rangle \rangle, s'$ 
66: end if
67: stop  $\langle \rangle, s$ 

```

D Formal Model of SPRESSO

We here present the full details of our formal model of SPRESSO. For our analysis regarding our authentication and privacy properties below, we will further restrict this generic model to suit the setting of the respective analysis.

We model SPRESSO as a web system (in the sense of Appendix A.3). We call a web system $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ an *SPRESSO web system* if it is of the form described in what follows.

D.1 Outline

The system $\mathcal{W} = \text{Hon} \cup \text{Web} \cup \text{Net}$ consists of web attacker processes (in Web), network attacker processes (in Net), a finite set FWD of forwarders, a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set IDP of web servers for the identity providers, and a finite set DNS of DNS servers, with $\text{Hon} := \text{BURP} \cup \text{IDP} \cup \text{FWD} \cup \text{DNS}$. More details on the processes in \mathcal{W} are provided below. Figure 7 shows the set of scripts \mathcal{S} and their respective string representations that are defined by the mapping script. The set E^0 contains only the trigger events as specified in Appendix A.3.

| $s \in \mathcal{S}$ | script(s) |
|---------------------|-----------------|
| R^{att} | att_script |
| script_rp | script_rp |
| script_rp_redir | script_rp_redir |
| script_idp | script_idp |
| script_fwd | script_fwd |

Fig. 7. List of scripts in \mathcal{S} and their respective string representations.

This outlines \mathcal{WS} . We will now define the DY processes in \mathcal{WS} and their addresses, domain names, and secrets in more detail. The scripts are defined in detail in Appendix D.16.

D.2 Addresses and Domain Names

The set IPs contains for every web attacker in Web, every network attacker in Net, every relying party in RP, every identity provider in IDP, every forwarder in FWD, every DNS server in DNS, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every forwarder FWD, every relying party in RP, every identity provider in IDP, every web attacker in Web, and every network attacker in Net. Browsers (in B) and DNS servers (in DNS) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

D.3 Keys and Secrets

The set \mathcal{N} of nonces is partitioned into four sets, an infinite sequence N , an infinite set K_{SSL} , an infinite set K_{sign} , and a finite set Secrets. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot{\cup} \underbrace{K_{\text{SSL}}}_{\text{finite}} \dot{\cup} \underbrace{K_{\text{sign}}}_{\text{finite}} \dot{\cup} \underbrace{\text{Secrets}}_{\text{finite}} .$$

The set N contains the nonces that are available for each DY process in \mathcal{W} (it can be used to create a run of \mathcal{W}).

The set K_{SSL} contains the keys that will be used for SSL encryption. Let $\text{sslkey}: \text{Doms} \rightarrow K_{\text{SSL}}$ be an injective mapping that assigns a (different) private key to every domain.

The set K_{sign} contains the keys that will be used by IdPs for signing IAs. Let $\text{signkey}: \text{IdPs} \rightarrow K_{\text{sign}}$ be an injective mapping that assigns a (different) private key to every identity provider.

The set Secrets is the set of passwords (secrets) the browsers share with the identity providers.

D.4 Identities

Identities are email addresses, which consist of a user name and a domain part. For our model, this is defined as follows:

Definition 42. An identity (email address) i is a term of the form $\langle name, domain \rangle$ with $name \in \mathcal{S}$ and $domain \in \text{Doms}$.

Let ID be the finite set of identities. By ID^y we denote the set $\{\langle name, domain \rangle \in ID \mid domain \in \text{dom}(y)\}$.

We say that an ID is governed by the DY process to which the domain of the ID belongs. Formally, we define the mapping $\text{governor} : ID \rightarrow \mathcal{W}$, $\langle name, domain \rangle \mapsto \text{dom}^{-1}(domain)$.

The governor of an ID will usually be an IdP, but could also be the attacker.

By $\text{secretOfID} : ID \rightarrow \text{Secrets}$ we denote the bijective mapping that assigns secrets to all identities.

Let $\text{ownerOfSecret} : \text{Secrets} \rightarrow \mathcal{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret.

Now, we define the mapping $\text{ownerOfID} : ID \rightarrow \mathcal{B}$, $i \mapsto \text{ownerOfSecret}(\text{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).

D.5 Tags and Identity Assertions

Definition 43. A tag is a term of the form $\text{enc}_s(\langle o, n \rangle, k)$ for some domain d , a nonce $n \in \mathcal{N}$, and a nonce (here used as a symmetric key) k .

Definition 44. An identity assertion (IA) is a term of the form $\text{sig}(\langle t, e, d' \rangle, k)$ for a tag t , an email address (identity) e , a domain d' and a nonce k . We call it an encrypted identity assertion (EIA) if it is additionally (symmetrically) encrypted (i.e., it is of the form $\text{enc}_s(s, k')$ if s is an IA and k' is a nonce).

D.6 Corruption

RPs, IdPs and FWDs can become corrupted: If they receive the message CORRUPT, they start collecting all incoming messages in their state and (upon triggering) send out all messages that are derivable from their state and collected input messages, just like the attacker process. We say that an RP, an IdP or an forwarder is *honest* if the according part of their state ($s.\text{corrupt}$) is \perp , and that they are corrupted otherwise.

We are now ready to define the processes in \mathcal{W} as well as the scripts in \mathcal{S} in more detail.

D.7 Processes in \mathcal{W} (Overview)

We first provide an overview of the processes in \mathcal{W} . All processes in \mathcal{W} (except for DNS servers) 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 $p \in \text{Hon} \cup \text{Web}$.

Web Attackers. Each $wa \in \text{Web}$ is a web attacker (see Appendix A.3), who uses only his own addresses for sending and listening.

Network Attackers. Each $na \in \text{Net}$ is a network attacker (see Appendix A.3), who uses all addresses for sending and listening.

Browsers. Each $b \in \mathcal{B}$ is a web browser as defined in Appendix C. The initial state contains all secrets owned by b , stored under the origin of the respective IdP. See Appendix D.11 for details.

Relying Parties. A relying party $r \in \text{RP}$ is a web server. RP knows four distinct paths: $/$, where it serves the index web page (`script_rp`), `/startLogin`, where it only accepts POST requests and mainly issues a fresh RP nonce (details see below), `/redirect`, where it only accepts requests with a valid login session token and serves `script_rp_redirect` to redirect the browser to the IdP, and `/login`, where it also only accepts POST requests with login data obtained during the login process by `script_rp` running in the browser. It checks this data and, if the data is deemed “valid”, it issues a service token (again, for details, see below). The RP keeps a list of such tokens in its state. Intuitively, a client having

such a token can use the service of the RP (for a specific identity record along with the token). Just like IdPs, RPs can become corrupted.

Identity Providers. Each IdP is a web server. As outlined in Section 2.1, users can authenticate to the IdP with their credentials. IdP tracks the state of the users with sessions. Authenticated users can receive IAs from the IdP. When receiving a special message (CORRUPT) IdPs can become corrupted. Similar to the definition of corruption for the browser, IdPs then start sending out all messages that are derivable from their state.

Forwarders. FWDs are web servers that have only one state and only serve the script `script_fwd`. See Appendix D.14 for details.

DNS. Each $dns \in \text{DNS}$ is a DNS server as defined in Appendix B.7. Their state contains the allocation of domain names to IP addresses.

D.8 SSL Key Mapping

Before we define the atomic DY processes in more detail, we first define the common data structure that holds the mapping of domain names to public SSL keys: For an atomic DY process p we define

$$sslkeys^p = \langle \{ \langle d, sslkey(d) \rangle \mid d \in \text{dom}(p) \} \rangle.$$

D.9 Web Attackers

Each $wa \in \text{Web}$ is a web attacker. The initial state of each wa is $s_0^{wa} = \langle attdoms, sslkeys, signkeys \rangle$, where $attdoms$ is a sequence of all domains along with the corresponding private keys owned by wa , $sslkeys$ is a sequence of all domains and the corresponding public keys, and $signkeys$ is a sequence containing all public signing keys for all IdPs. All other parties use the attacker as a DNS server.

D.10 Network Attackers

As mentioned, each network attacker na is modeled to be a network attacker as specified in Appendix A.3. We allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \text{IPs}$. The initial state is $s_0^{na} = \langle attdoms, sslkeys, signkeys \rangle$, where $attdoms$ is a sequence of all domains along with the corresponding private keys owned by the attacker na , $sslkeys$ is a sequence of all domains and the corresponding public keys, and $signkeys$ is a sequence containing all public signing keys for all IdPs.

D.11 Browsers

Each $b \in \text{B}$ is a web browser as defined in Appendix C, with $I^b := \text{addr}(b)$ being its addresses.

To define the initial state, first let $ID^b := \text{ownerOfID}^{-1}(b)$ be the set of all IDs of b , $ID^{b,d} := \{i \mid \exists x: i = \langle x, d \rangle \in ID^b\}$ be the set of IDs of b for a domain d , and $\text{SecretDomains}^b := \{d \mid ID^{b,d} \neq \emptyset\}$ be the set of all domains that b owns identities for.

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (ssl) key, according to the mapping $sslkey$; the DNS address is $\text{addr}(p)$ with $p \in \mathcal{W}$; the list of secrets contains an entry $\langle \langle d, \mathcal{S} \rangle, s \rangle$ for each $d \in \text{SecretDomains}^b$ and $s = \text{secretOfID}(i)$ for some $i \in ID^{b,d}$ (s is the same for all i); ids is $\langle ID^b \rangle$; sts is empty.

D.12 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) with the addresses $I^r := \text{addr}(r)$. Its initial state s_0^r contains its domains, the private keys associated with its domains, the DNS server address, and the domain name of a forwarder. The full state additionally contains the sets of service tokens and login session identifiers the RP has issued. RP only accepts HTTPS requests.

RP manages two kinds of sessions: The *login sessions*, which are only used during the login phase of a user, and the *service sessions* (we call the session identifier of a service session a *service token*). Service sessions allow a user to use RP's services. The ultimate goal of a login flow is to establish such a service session.

In a typical flow with one client, r will first receive an HTTP GET request for the path $/$. In this case, r returns the script `script_rp` (see below).

After the user entered her email address, r will receive an HTTPS POST XMLHTTPRequest for the path `/startLogin`. In this request, it expects the email address the user entered. The relying party then contacts the user's email provider to retrieve the SPRESSO support document (where it extracts the public key of the IdP). After that, r selects the nonces $rpNonce$, $iaKey$, $tagKey$, and $loginSessionToken$. It creates the tag as the (symmetric) encryption of its own domain and the $rpNonce$ with $tagKey$. It then returns to the browser the $loginSessionToken$, the $tagKey$, and the domain of the forwarder ($S(r).FWDDomain$).

When the RP document in the browser opens the login dialog, r receives a third request, in this case a GET request for the path `/redir` with a parameter containing $loginSessionToken$. This is now used by r to look up the user's session and redirect the user to the IdP (this redirection serves mainly to hide the referer string from the request to IdP). For this, r sends the script `script_rp_redir` and, in its initial script state, defines that the script should redirect the user to the URL of the login dialog (which is "https://" plus the domain of the user's email address plus `/.well-known/spresso-login`).

Finally, r receives a last request in the login flow. This POST request contains the encrypted IA and the $loginSessionToken$. To conclude the login, r looks up the user's login session, decrypts the IA, and checks that it is a signature over the tag, the user's email address, and the FWD domain. If successful, r returns a new service token, which is also stored in the state of r .

If r receives a corrupt message, it becomes corrupt and acts like the attacker from then on.

We now provide the formal definition of r as an atomic DY process (I^r, Z^r, R^r, s_0^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 45. A login session record is a term of the form $\langle email, rpNonce, iaKey, tag \rangle$ with $email \in \text{ID}$ and $rpNonce, iaKey, tag \in \mathcal{N}$.

Definition 46. A state $s \in Z^r$ of an RP r is a term of the form $\langle \text{DNSAddress}, \text{FWDDomain}, \text{keyMapping}, \text{sslkeys}, \text{pendingDNS}, \text{pendingRequests}, \text{loginSessions}, \text{serviceTokens}, \text{wkCache}, \text{corrupt} \rangle$ where $\text{DNSAddress} \in \text{IPs}$, $\text{FWDDomain} \in \text{Doms}$, $\text{keyMapping} \in [\mathbb{S} \times \mathcal{N}]$, $\text{sslkeys} = \text{sslkeys}^r$, $\text{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{serviceTokens} \in [\mathcal{N} \times \mathbb{S}]$, $\text{loginSessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary of login session records, $\text{wkCache} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $\text{corrupt} \in \mathcal{T}_{\mathcal{N}}$.

The initial state s_0^r of r is a state of r with $s_0^r.\text{serviceTokens} = s_0^r.\text{loginSessions} = s_0^r.\text{wkCache} = \langle \rangle$, $s_0^r.\text{corrupt} = \perp$, and $s_0^r.\text{keyMapping}$ is the same as the keymapping for browsers above.

We now specify the relation R^r . Just like in Appendix C, we describe this relation by a non-deterministic algorithm.

Algorithm 8 Sending the response to a startLogin XMLHTTPRequest

```

1: function SENDSTARTLOGINRESPONSE( $a, f, k, n, email, inDomain, s'$ )
2:   let  $rpNonce := \nu_1$ 
3:   let  $tagKey := \nu_2$ 
4:   let  $iaKey := \nu_3$ 
5:   let  $loginSessionToken := \nu_4$ 
6:   let  $tag := \text{enc}_s(\langle inDomain, rpNonce \rangle, tagKey)$ 
7:   let  $s'.\text{loginSessions}[loginSessionToken] := \langle email, rpNonce, iaKey, tag \rangle$ 
8:   let  $body := \langle \langle tagKey, tagKey \rangle \langle loginSessionToken, loginSessionToken \rangle, \langle FWDDomain, s'.FWDDomain \rangle \rangle$ 
9:   let  $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, body \rangle, k)$ 
10:  stop  $\langle \langle f, a, m' \rangle \rangle, s'$ 
11: end function

```

Algorithm 9 Relation of a Relying Party R^r

Input: $\langle a, f, m \rangle, s$
1: **if** $s'.\text{corrupt} \neq \perp \vee m \equiv \text{CORRUPT}$ **then**

```

2:   let  $s'.\text{corrupt} := \langle \langle a, f, m \rangle, s'.\text{corrupt} \rangle$ 
3:   let  $m' \leftarrow d_V(s')$ 
4:   let  $a' \leftarrow \text{IPs}$ 
5:   stop  $\langle \langle a', a, m' \rangle, s' \rangle$ 
6: end if
7: if  $\exists \langle \text{reference}, \text{request}, \text{key}, f \rangle \in s'.\text{pendingRequests}$ 
    $\hookrightarrow$  such that  $\pi_1(\text{dec}_s(m, \text{key})) \equiv \text{HTTPResp}$  then ▷ Encrypted HTTP response
8:   let  $m' := \text{dec}_s(m, \text{key})$ 
9:   if  $m'.\text{nonce} \neq \text{request.nonce}$  then
10:    stop  $\langle \rangle, s$ 
11:   end if
12:   remove  $\langle \text{reference}, \text{request}, \text{key}, f \rangle$  from  $s'.\text{pendingRequests}$ 
13:   let  $a', f', k, n, \text{email}, \text{inDomain}$  such that  $\langle a', f', k, n, \text{email}, \text{inDomain} \rangle \equiv \text{reference}$  if possible; otherwise stop  $\langle \rangle, s$ 
14:   let  $s'.\text{wkCache}[\text{request.host}] := m'.\text{body}$ 
15:   SENDSTARTLOGINRESPONSE( $a', f', k, n, \text{email}, \text{inDomain}, s'$ )
16: else if  $m \in \text{DNSResponses}$  then ▷ Successful DNS response
17:   if  $m.\text{nonce} \notin s.\text{pendingDNS} \vee m.\text{result} \notin \text{IPs} \vee m.\text{domain} \neq \pi_2(s.\text{pendingDNS}).\text{host}$  then
18:     stop  $\langle \rangle, s$ 
19:   end if
20:   let  $\langle \text{reference}, \text{message} \rangle := s.\text{pendingDNS}[m.\text{nonce}]$ 
21:   let  $s'.\text{pendingRequests} := s'.\text{pendingRequests}$ 
      $\hookrightarrow + \langle \text{reference}, \text{message}, \nu_5, m.\text{result} \rangle$ 
22:   let  $\text{message} := \text{enc}_a(\langle \text{message}, \nu_5 \rangle, s'.\text{keyMapping}[\text{message.host}])$ 
23:   let  $s'.\text{pendingDNS} := s'.\text{pendingDNS} - m.\text{nonce}$ 
24:   stop  $\langle \langle m.\text{result}, a, \text{message} \rangle, s' \rangle$ 
25: else ▷ Handle HTTP requests
26:   let  $m_{\text{dec}}, k, k', \text{inDomain}$  such that
      $\hookrightarrow \langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in s.\text{sslkeys}$ 
      $\hookrightarrow$  if possible; otherwise stop  $\langle \rangle, s$ 
27:   let  $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$  such that
      $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$ 
      $\hookrightarrow$  if possible; otherwise stop  $\langle \rangle, s$ 
28:   if  $\text{path} \equiv /$  then ▷ Serve index page.
29:     let  $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, \langle \text{script\_rp}, \text{initState\_rp} \rangle, k \rangle, k)$  ▷ Initial state defined for script_rp (below).
30:     stop  $\langle \langle f, a, m' \rangle, s' \rangle$ 
31:   else if  $\text{path} \equiv /startLogin \wedge \text{method} \equiv \text{POST}$  then ▷ Serve start login request.
32:     if  $\text{body} \notin \text{ids}$  then
33:       stop  $\langle \rangle, s$ 
34:     end if
35:     let  $\text{domain} := \text{body.domain}$ 
36:     if  $\text{domain} \in s.\text{wkCache}$  then
37:       SENDSTARTLOGINRESPONSE( $a, f, k, n, \text{body}, \text{inDomain}, s'$ )
38:     else
39:       let  $\text{message} := \langle \text{HTTPReq}, \nu_6, \text{GET}, \text{domain}, /.well-known/spresso-info, \langle \rangle, \langle \rangle, \langle \rangle \rangle$ 
40:       let  $s'.\text{pendingDNS}[\nu_6] := \langle \langle a, f, k, n, \text{body}, \text{inDomain} \rangle, \text{message} \rangle$ 
41:       stop  $\langle \langle s'.\text{DNSaddress}, a, \langle \text{DNSResolve}, \text{domain}, \nu_6 \rangle \rangle, s' \rangle$ 
42:     end if
43:   else if  $\text{path} \equiv /redir \wedge \text{method} \equiv \text{GET}$  then ▷ Serve redirection script.
44:     let  $\text{loginSession} := s'.\text{loginSessions}[\text{body}[\text{loginSessionToken}]]$ 
45:     if  $\text{loginSession} \equiv \langle \rangle$  then
46:       stop  $\langle \rangle, s$ 
47:     end if
48:     let  $\text{domain} := \text{loginSession.email.domain}$ 
49:     let  $\text{params} := \langle \langle \text{email}, \text{loginSession.email} \rangle, \langle \text{tag}, \text{loginSession.tag} \rangle, \langle \text{iaKey}, \text{loginSession.iaKey} \rangle, \langle \text{FWDDomain}, s'.\text{FWDDomain} \rangle \rangle$ 
      $\hookrightarrow \langle \text{iaKey}, \text{loginSession.iaKey} \rangle, \langle \text{FWDDomain}, s'.\text{FWDDomain} \rangle$ 
50:     let  $\text{url} := \langle \text{URL}, S, \text{domain}, /.well-known/spresso-login, \text{params} \rangle$ 

```

```

51:   let  $m'$  := encs(⟨HTTPResp, n, 200, ⟨⟩, ⟨script_rp_redir, url⟩⟩, k)
52:   stop ⟨⟨f, a, m'⟩⟩, s'
53:   else if path ≡ /login ∧ method ≡ POST then ▷ Serve login request.
54:     if headers[Origin] ≠ ⟨inDomain, S⟩ ∨ body[loginSessionToken] ≡ ⟨⟩ then
55:       stop ⟨⟩, s
56:     end if
57:     let loginSession := s'.loginSessions[body[loginSessionToken]]
58:     if loginSession ≡ ⟨⟩ then
59:       stop ⟨⟩, s
60:     end if
61:     let s'.loginSessions := s'.loginSessions - body[loginSessionToken]
62:     let ia := decs(body[eia], loginSession.iaKey)
63:     let e := ⟨loginSession.tag, loginSession.email, s'.FWDDomain⟩
64:     if checksig(e, ia, s'.wkCache[loginSession.email.domain][signkey]) ≡ ⊥ then
65:       stop ⟨⟩, s
66:     end if
67:     let serviceTokenNonce := ν7
68:     let serviceToken := ⟨serviceTokenNonce, loginSession.email⟩
69:     let s'.serviceTokens := s'.serviceTokens + ⟨⟩ serviceToken
70:     let  $m'$  := encs(⟨HTTPResp, n, 200, ⟨⟩, serviceToken⟩, k)
71:     stop ⟨⟨f, a, m'⟩⟩, s'
72:   end if
73: end if
74: stop ⟨⟩, s

```

D.13 Identity Providers

An identity provider $i \in \text{IdPs}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the addresses $I^i := \text{addr}(i)$. Its initial state s_0^i contains a list of its domains and (private) SSL keys, a list of users and identities, and a private key for signing UCs. Besides this, the full state of i further contains a list of used nonces, and information about active sessions.

IdPs react to three types of requests:

First, they provide the “well-known document”, a machine-readable document which contains the IdP’s verification key. This document is served upon a GET request to the path `/well-known/spresso-info`.

Second, upon a request to the LD path (i.e., `/well-known/spresso-login`), an IdP serves the login dialog script, i.e., `script_idp`. Into the initial state of this script, IdPs encode whether the browser is already logged in or not. Further, IdP issues an XSRF token to the browser (in the same way RPs do).

The login dialog will eventually send an XMLHTTPRequest to the path `loginxhr`, where it retrieves the IA. This is also the last type of requests IdPs answer to. Before serving the response to this request, IdP checks whether the user is properly authenticated. It then creates the IA and sends it to the browser.

Formal description. In the following, we will first define the (initial) state of i formally and afterwards present the definition of the relation R^i .

To define the initial state, we will need a term that represents the “user database” of the IdP i . We will call this term $userSet^i$. This database defines, which secret is valid for which identity. It is encoded as a mapping of identities to secrets. For example, if the secret $secret_1$ is valid for the identities id_1 and the secret $secret_2$ is valid for the identity id_2 , the $userSet^i$ looks as follows:

$$userSet^i = [id_1:secret_1, id_2:secret_2]$$

We define $userSet^i$ as $userSet^i = \langle \{ \langle u, \text{secretOfID}(u) \rangle \mid u \in \text{ID}^i \} \rangle$.

Definition 47. A state $s \in Z^i$ of an IdP i is a term of the form $\langle \text{sslkeys}, \text{users}, \text{signkey}, \text{sessions}, \text{corrupt} \rangle$ where $\text{sslkeys} = \text{sslkeys}^i$, $\text{users} = userSet^i$, $\text{signkey} \in \mathcal{N}$ (the key used by the IdP i to sign UCs), $\text{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\text{corrupt} \in \mathcal{T}_{\mathcal{N}}$.

An initial state s_0^i of i is a state of the form $\langle \text{sslkeys}^i, \text{userSet}^i, \text{signkey}(i), \langle \rangle, \perp \rangle$.

The relation R^i that defines the behavior of the IdP i is defined as follows:

Algorithm 10 Relation of IdP R^i

Input: $\langle a, f, m \rangle, s$

- 1: **let** $s' := s$
- 2: **if** $s'.\text{corrupt} \neq \perp \vee m \equiv \text{CORRUPT}$ **then**
- 3: **let** $s'.\text{corrupt} := \langle \langle a, f, m \rangle, s'.\text{corrupt} \rangle$
- 4: **let** $m' \leftarrow d_V(s')$
- 5: **let** $a' \leftarrow \text{IPs}$
- 6: **stop** $\langle \langle a', a, m' \rangle \rangle, s'$
- 7: **end if**
- 8: **let** $m_{\text{dec}}, k, k', \text{inDomain}$ **such that**
- $\hookrightarrow \langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in s.\text{sslkeys}$
- \hookrightarrow **if possible; otherwise stop** $\langle \rangle, s$
- 9: **let** $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$ **such that**
- $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$
- \hookrightarrow **if possible; otherwise stop** $\langle \rangle, s$
- 10: **if** $\text{path} \equiv /.well-known/spresso-info$ **then** ▷ Serve support document.
- 11: **let** $\text{wkDoc} := \langle \langle \text{signkey}, \text{pub}(s'.\text{signkey}) \rangle \rangle$
- 12: **let** $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, \text{wkDoc} \rangle, k)$
- 13: **stop** $\langle \langle f, a, m' \rangle \rangle, s'$
- 14: **else if** $\text{path} \equiv /.well-known/spresso-login$ **then** ▷ Serve login dialog.
- 15: **let** $\text{sessionid} := \text{headers}[\text{Cookie}][\text{sessionid}]$
- 16: **let** $\text{email} := s'.\text{sessions}[\text{sessionid}]$
- 17: **let** $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, \langle \text{script_idp}, \langle \text{start}, \text{email}, \langle \rangle \rangle \rangle \rangle, k)$ ▷ Initial scriptstate of script_idp (defined below).
- 18: **stop** $\langle \langle f, a, m' \rangle \rangle, s'$
- 19: **else if** $\text{path} \equiv /sign \wedge \text{method} \equiv \text{POST}$ **then** ▷ Serve signing request.
- 20: **let** $\text{sessionid} := \text{headers}[\text{Cookie}][\text{sessionid}]$
- 21: **let** $\text{loggedIns} := s'.\text{sessions}[\text{sessionid}]$
- 22: **if** $\text{body}[\text{email}] \neq \text{loggedIns} \wedge \text{body}[\text{password}] \neq s'.\text{userSet}[\text{body}[\text{email}]]$ **then**
- 23: **stop** $\langle \rangle, s$
- 24: **end if**
- 25: **let** $ia := \text{sig}(\langle \text{body}[\text{tag}], \text{body}[\text{email}], \text{body}[\text{FWDDomain}] \rangle, s'.\text{signkey})$
- 26: **let** $\text{sessionid} := \nu_8$
- 27: **let** $s'.\text{sessions}[\text{sessionid}] := \text{body}[\text{email}]$
- 28: **let** $\text{setCookie} := \langle \text{Set-Cookie}, \langle \langle \text{sessionid}, \text{sessionid}, \top, \top, \top \rangle \rangle \rangle$
- 29: **let** $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \text{setCookie} \rangle, ia \rangle, k)$
- 30: **stop** $\langle \langle f, a, m' \rangle \rangle, s'$
- 31: **end if**
- 32: **stop** $\langle \rangle, s$

D.14 Forwarders

We define FWDs formally as atomic DY processes $\text{fwd} = (I^{\text{fwd}}, Z^{\text{fwd}}, R^{\text{fwd}}, s_0^{\text{fwd}})$. As already mentioned, we define $I^{\text{fwd}} = \text{addr}(\text{fwd})$ with the set of states Z^{fwd} being all terms of the form $\langle \text{sslkeys}, \text{corrupt} \rangle$ for $\text{sslkeys}, \text{corrupt} \in \mathcal{T}_{\mathcal{N}_C}$.

The initial state s_0^{fwd} of an FWD contains the private key of its domain and the corruption state: $s_0^{\text{fwd}} = \langle \text{sslkeys}^{\text{fwd}}, \perp \rangle$.

An FWD responds to any HTTPS request with script_fwd and its initial state, which is empty.

We now specify the relation R^{fwd} of FWDs. We describe this relation by a non-deterministic algorithm.

Algorithm 11 Relation of an FWD R^{fwd}

Input: $\langle a, f, m \rangle, s$

- 1: **if** $s.\text{corrupt} \neq \perp \vee m \equiv \text{CORRUPT}$ **then**

```

2:   let  $s'.\text{corrupt} := \langle\langle a, f, m \rangle, s.\text{corrupt}\rangle$ 
3:   let  $m' \leftarrow d_V(s')$ 
4:   let  $a' \leftarrow \text{IPs}$ 
5:   stop  $\langle\langle a', a, m' \rangle\rangle, s'$ 
6: end if
7: let  $m_{\text{dec}}, k, k', \text{inDomain}$  such that
    $\hookrightarrow \langle m_{\text{dec}}, k \rangle \equiv \text{dec}_a(m, k') \wedge \langle \text{inDomain}, k' \rangle \in s$ 
    $\hookrightarrow$  if possible; otherwise stop  $\langle \rangle, s$ 
8: let  $n, \text{method}, \text{path}, \text{parameters}, \text{headers}, \text{body}$  such that
    $\hookrightarrow \langle \text{HTTPReq}, n, \text{method}, \text{inDomain}, \text{path}, \text{parameters}, \text{headers}, \text{body} \rangle \equiv m_{\text{dec}}$ 
    $\hookrightarrow$  if possible; otherwise stop  $\langle \rangle, s$ 
9: let  $m' := \text{enc}_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, \langle \text{script\_fwd}, \langle \rangle \rangle, k)$ 
10: stop  $\langle\langle f, a, m' \rangle\rangle, s$ 

```

D.15 DNS Servers

As already outlined above, DNS servers are modeled as generic DNS servers presented in Appendix B.7. Their (static) state is set according to the allocation of domain names to IP addresses. DNS servers may not become corrupted.

D.16 SPRESSO Scripts

As already mentioned in Appendix D.1, the set \mathcal{S} of the web system $\mathcal{SWS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ consists of the scripts R^{att} , script_rp , script_idp , and script_fwd , with their string representations being att_script , script_rp , script_idp , and script_fwd (defined by script).

In what follows, the scripts script_rp , script_idp , and script_fwd are defined formally. First, we introduce some notation and helper functions.

Notations and Helper Functions. In the formal description of the scripts we use an abbreviation for URLs. We write $\text{URL}_{\text{path}}^d$ to describe the following URL term: $\langle \text{URL}, \mathcal{S}, d, \text{path}, \langle \rangle \rangle$.

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

CHOOSEINPUT.

The state of a document contains a term scriptinputs which records the input this document has obtained so far (via XHRs and postMessages, append-only). If the script of the document is activated, it will typically need to pick one input message from the sequence scriptinputs and record which input it has already processed. For this purpose, the function $\text{CHOOSEINPUT}(\text{scriptinputs}, \text{pattern})$ is used. If called, it chooses the first message in scriptinputs that matches pattern and returns it.

Algorithm 12 Choose an unhandled input message for a script

```

1: function CHOOSEINPUT( $\text{scriptinputs}, \text{pattern}$ )
2:   let  $i$  such that  $i = \min\{j : \pi_j(\text{scriptinputs}) \sim \text{pattern}\}$  if possible; otherwise return  $\perp$ 
3:   return  $\pi_i(\text{scriptinputs})$ 
4: end function

```

PARENTWINDOW. To determine the nonce referencing the active document in the parent window in the browser, the function $\text{PARENTWINDOW}(\text{tree}, \text{docnonce})$ 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 active document in 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) or no active document, PARENTWINDOW returns docnonce .

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 $\text{SUBWINDOWS}(\text{tree}, \text{docnonce})$ returns $\langle \rangle$. Otherwise, let $\langle \text{docnonce}, \text{origin}, \text{script}, \text{scriptstate}, \text{scriptinputs}, \text{subwindows}, \text{active} \rangle$ denote the subterm of tree corresponding to the document referred to by docnonce . Then, $\text{SUBWINDOWS}(\text{tree}, \text{docnonce})$ 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 active document's nonce. If there is no such window or no active document, it returns $docnonce$.

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.

GETPARAMETERS. Works exactly as *GETORIGIN*, but returns the document's parameters instead.

Relying Party Index Page ($script_rp$). As defined in Appendix A.3, a script is a relation that takes as input a term and outputs a new term. As specified in Appendix C (Triggering the Script of a Document ($m = TRIGGER$, $action = 1$)) and formally specified in Algorithm 5, 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 Appendix C and Algorithm 5. The output term contains, among other information, the new internal $scriptstate$.

We first describe the structure of the internal $scriptstate$ of the script $script_rp$.

Definition 48. A $scriptstate$ s of $script_rp$ is a term of the form $\langle q, loginSessionToken, refXHR, tagKey, FWDDomain \rangle$ where $q \in \mathbb{S}$, $loginSessionToken, refXHR, tagKey \in \mathcal{N} \cup \{\perp\}$, $FWDDomain \in \mathcal{T}_{\mathcal{N}}$.

The initial $scriptstate$ $initState_{rp}$ of $script_rp$ is $\langle start, \perp, \perp, \perp, \perp \rangle$.

We now specify the relation $script_rp$ formally. We describe this relation by a non-deterministic algorithm.

Just like all scripts, as explained in Appendix C (see also Algorithm 5 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 $scriptstate$ (as defined in Definition 48), a sequence of all inputs $scriptinputs$ (also containing already handled inputs), a dictionary $cookies$ of all accessible cookies of the document's domain, the local storage $localStorage$ 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 $cookies'$, a new local storage $localStorage'$, and a term $command$ denoting a command to the browser.

Algorithm 13 Relation of $script_rp$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, \rightarrow ids, secret \rangle$

- 1: **let** $s' := scriptstate$
- 2: **let** $command := \langle \rangle$
- 3: **let** $origin := GETORIGIN(tree, docnonce)$
- 4: **switch** $s'.q$ **do**
- 5: **case** $start$
- 6: **let** $s'.email \leftarrow ids$
- 7: **let** $s'.refXHR := \lambda_1$
- 8: **let** $command := \langle XMLHTTPREQUEST, URL_{/startLogin}^{origin.domain}, POST, s'.email, s'.refXHR \rangle$
- 9: **let** $s'.q := expectStartLoginResponse$
- 10: **case** $expectStartLoginResponse$
- 11: **let** $pattern := \langle XMLHTTPREQUEST, *, s'.refXHR \rangle$


```

12:   let input := CHOOSEINPUT(scriptinputs.pattern)
13:   if input ≠ ⊥ then
14:     let s'.loginSessionToken := π2(input)[loginSessionToken]
15:     let s'.tagKey := π2(input)[tagKey]
16:     let s'.FWDDomain := π2(input)[FWDDomain]
17:     let command :=
18:       ↪ ⟨HREF, URL, S, origin.domain, /redir, ⟨⟨loginSessionToken, s'.loginSessionToken⟩⟩, _BLANK, ⟨⟩⟩
19:     let s'.q := expectFWDReady
20:   end if
21: case expectFWDReady
22:   let fwdWindowNonce := SUBWINDOWS(tree, AUXWINDOW(tree, docnonce)).l.nonce
23:   let pattern := ⟨POSTMESSAGE, fwdWindowNonce, ⟨s'.FWDDomain, S⟩, ready⟩
24:   let input := CHOOSEINPUT(scriptinputs.pattern)
25:   if input ≠ ⊥ then
26:     let command := ⟨POSTMESSAGE, fwdWindowNonce, ⟨tagKey, tagKey⟩, ⟨s'.FWDDomain, S⟩⟩
27:     let s'.q := expectEIA
28:   end if
29: case expectEIA
30:   let fwdWindowNonce := SUBWINDOWS(tree, AUXWINDOW(tree, docnonce)).l.nonce
31:   let pattern := ⟨POSTMESSAGE, fwdWindowNonce, ⟨s'.FWDDomain, S⟩, ⟨eia, *⟩⟩
32:   let input := CHOOSEINPUT(scriptinputs.pattern)
33:   if input ≠ ⊥ then
34:     let eia := π2(π4(input))
35:     let s'.refXHR := λ1
36:       ↪ ⟨XMLHTTPREQUEST, URL/loginorigin.domain, POST, body, s'.refXHR⟩
37:     let s'.q := expectServiceToken
38:   end if
39: stop ⟨s', cookies, localStorage, sessionStorage, command⟩

```

Relying Party Redirection Page (script_rp_redir). This simple script (which is loaded from RP in a regular run) is used to redirect the login dialog window to the actual login dialog (IdPdoc) loaded from IdP. It expects the URL of the page to which the browser should be redirected in its initial (and only) state.

Algorithm 14 Relation of *script_rp_redir*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, \dots \rangle$
 $\hookrightarrow ids, secret$

- 1: let $command := \langle HREF, scriptstate, \perp, \top \rangle$
- 2: stop $\langle scriptstate, cookies, localStorage, sessionStorage, command \rangle$

Login Dialog Script (script_idp). This script models the contents of the login dialog.

Definition 49. A scriptstate s of *script_idp* is a term of the form $\langle q, email \rangle$ with $q \in \mathbb{S}$, $email \in ID \cup \{\langle \rangle\} \in \mathcal{T}$. We call the scriptstate s an initial scriptstate of *script_idp* iff $s \sim \langle start, * \rangle$.

We now formally specify the relation *script_idp* of the LD's scripting process.

Algorithm 15 Relation of *script_idp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, \dots \rangle$
 $\hookrightarrow ids, secret$

- 1: let $s' := scriptstate$
- 2: let $command := \langle \rangle$
- 3: let $origin := GETORIGIN(tree, docnonce)$
- 4: switch $s'.q$ do
- 5: case start

```

6:   let email := GETPARAMETERS(tree, docnonce)[email]
7:   let tag := GETPARAMETERS(tree, docnonce)[tag]
8:   let FWDDomain := GETPARAMETERS(tree, docnonce)[FWDDomain]
9:   let body := ⟨⟨email, email⟩, ⟨password, secret⟩, ⟨tag, tag⟩, ⟨FWDDomain, FWDDomain⟩⟩
10:  let command := ⟨XMLHTTPREQUEST, URLorigin.domain, POST, body, ⊥⟩
11:  let s'.q := expectIA
12:  case expectIA
13:  let pattern := ⟨XMLHTTPREQUEST, *, *⟩
14:  let input := CHOOSEINPUT(scriptinputs, pattern)
15:  if input ≠ ⊥ then
16:    let iaKey := GETPARAMETERS(tree, docnonce)[iaKey]
17:    let FWDDomain := GETPARAMETERS(tree, docnonce)[FWDDomain]
18:    let tag := GETPARAMETERS(tree, docnonce)[tag]
19:    let eia := encs(π2(input), iaKey)
20:    let url := ⟨URL, S, FWDDomain, /, ⟨⟨tag, tag⟩, ⟨eia, eia⟩⟩⟩
21:    let command := ⟨IFRAME, url, _SELF⟩
22:    let s'.q := stop
23:  end if
24: stop ⟨s', cookies, localStorage, sessionStorage, command⟩

```

Forwarder Script (script_fwd).

Definition 50. A scriptstate s of $script_fwd$ is a term of the form q with $q \in \mathbb{S}$. We call s the initial scriptstate of $script_fwd$ iff $s \equiv \text{start}$.

We now formally specify the relation $script_rp_index$ of the FWD's scripting process.

Algorithm 16 Relation of $script_fwd$

Input: $(tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage,$
 $\hookrightarrow ids, secret)$

```

1: let s' := scriptstate
2: let command := ⟨⟩
3: let target := OPENERWINDOW(tree, PARENTWINDOW(tree, docnonce))
4: switch s'.q do
5:   case start
6:     let command := ⟨POSTMESSAGE, target, ready, ⊥⟩
7:     let s'.q := expectTagKey
8:   case expectTagKey
9:     let pattern := ⟨POSTMESSAGE, target, *, ⟨tagKey, *⟩⟩
10:    let input := CHOOSEINPUT(scriptinputs, pattern)
11:    if input ≠ ⊥ then
12:      let tagKey := π2(π4(input))
13:      let tag := GETPARAMETERS(tree, docnonce)[tag]
14:      let eia := GETPARAMETERS(tree, docnonce)[eia]
15:      let rpOrigin := ⟨decs(tag, tagKey).1, S⟩
16:      let command := ⟨POSTMESSAGE, target, ⟨eia, eia⟩, rpOrigin⟩
17:      let s'.q := stop
18:    end if
19: stop ⟨s', cookies, localStorage, sessionStorage, command⟩

```

E Formal Security Properties Regarding Authentication

To state the security properties for SPRESSO, we first define an *SPRESSO web system for authentication analysis*. This web system is based on the SPRESSO web system and only considers one network attacker (which subsumes all web attackers and further network attackers).

Definition 51. Let $\mathcal{SWS}^{auth} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ an SPRESSO web system. We call \mathcal{SWS}^{auth} an SPRESSO web system for authentication analysis iff \mathcal{W} contains only one network attacker process `attacker` and no other attacker processes (i.e., $\text{Net} = \{\text{attacker}\}$, $\text{Web} = \emptyset$). Further, \mathcal{W} contains no DNS servers. DNS servers are assumed to be dishonest, and hence, are subsumed by `attacker`. In the initial state s_0^b of each browser b in \mathcal{W} , the DNS address is `addr(attacker)`. Also, in the initial state s_0^r of each relying party r , the DNS address is `addr(attacker)`.

The security properties for SPRESSO are formally 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 an HTTPS POST request m . We refer to m as the *request corresponding to $\langle n, i \rangle$* .

Definition 52. Let \mathcal{SWS}^{auth} be an SPRESSO web system for authentication analysis. We say that \mathcal{SWS}^{auth} is secure if for every run ρ of \mathcal{SWS}^{auth} , every state (S^j, E^j, N^j) in ρ , every $r \in \text{RP}$ that is honest in S^j with $s_0(r).\text{FWDDomain}$ being a domain of an FWD that is honest in S^j , every RP service token of the form $\langle n, i \rangle$ recorded in $S^j(r).\text{serviceTokens}$, the following two conditions are satisfied:

(A) If $\langle n, i \rangle$ is derivable from the attackers knowledge in S^j (i.e., $\langle n, i \rangle \in d_0(S^j(\text{attacker}))$), then it follows that the browser b owning i is fully corrupted in S^j (i.e., the value of `isCorrupted` is `FULLCORRUPT`) or `governor(i)` is not an honest IdP (in S^j).

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

F Proof of Theorem 2

Before we prove Theorem 2, we show some general properties of the \mathcal{SWS}^{auth} .

F.1 Properties of \mathcal{SWS}^{auth}

Let $\mathcal{SWS}^{auth} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ be a web system. In the following, we write $s_x = (S^x, E^x, N^x)$ for the states of a web system.

Definition 53. 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 $u \in \mathbb{N}$, a set of events E and some addresses x, y with $\langle x, y, m \rangle \in E$.

Definition 54. We say that a term t is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \rightarrow s_{i+1}$ of a run $\rho = (s_0, s_1, \dots)$) if t is derivable from t' with the knowledge available to P , i.e.,

$$t \in d_0(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))$$

Definition 55. We say that a set of processes P leaks a term t (in a processing step $s_i \rightarrow s_{i+1}$) to a set of processes P' if there exists a message m that is emitted (in $s_i \rightarrow s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \rightarrow s_{i+1}$. If we omit P' , we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 56. We say that an DY process p created a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p' .

Definition 57. 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 6).

Definition 58. In a similar fashion, we say that an RP r accepted a message (as a response to some request) if the RP decrypted the message (RPs can only accept HTTPS messages) and added the message's body to the `wkCache` in its state (i.e., Line 14 of Algorithm 9 was called).

Definition 59. We say that an atomic DY process p knows a term t in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_0(S(p))$.

Definition 60. We say that a script initiated a request r if a browser triggered the script (in Line 10 of Algorithm 5) 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.

For a run $\rho = (s_0, s_1, \dots)$ of \mathcal{SWS}^{auth} , we state the following lemmas:

Lemma 1. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of \mathcal{SWS}^{auth} an honest relying party r (I) emits an HTTPS request of the form

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

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and (II) in the initial state s_0 the private key k' is only known to u , and (III) u never leaks k' , then all of the following statements are true:

1. There is no state of \mathcal{SWS}^{auth} where any party except for u knows k' , thus no one except for u can decrypt req .
2. If there is a processing step $s_j \rightarrow s_{j+1}$ where the RP r leaks k to $\mathcal{W} \setminus \{u, r\}$ there is a processing step $s_h \rightarrow s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ or r is corrupted in s_j .
3. The value of the host header in req is the domain that is assigned the public key $\text{pub}(k')$ in RP's keymapping $s_0.\text{keyMapping}$ (in its initial state).
4. If r accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ and r is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ prior to s_j , then u created the HTTPS response m' to the HTTPS request m , i.e., the nonce of the HTTP request req is not known to any atomic process p , except for the atomic DY processes r and u .

Proof. (1) follows immediately from the condition. If k' is initially only known to u and u never leaks k' , i.e., even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u , k' cannot be known to any other party. Thus, nobody except for u can derive req from m .

(2) We assume that r leaks k to $\mathcal{W} \setminus \{u, r\}$ in the processing step $s_j \rightarrow s_{j+1}$ without u prior leaking the key k to anyone except for u and r and that the RP is not fully corrupted in s_j , and lead this to a contradiction.

The RP is honest in s_j . From the definition of the RP, we see that the key k is always a fresh nonce that is not used anywhere else. Further, the key is stored in `pendingRequests`. The information from `pendingRequests` is not extracted or used anywhere else, except when handling the received messages, where it is only checked against. Hence, r does not leak k to any other party in s_j (except for u and r). This proves (2).

(3) Per the definition of RPs (Algorithm 9), a host header is always contained in HTTP requests by RPs. From Line 22 of Algorithm 9 we can see that the encryption key for the request req was chosen using the host header of the message. It is chosen from the `keyMapping` in RP's state, which is never changed during ρ . This proves (3).

(4) An HTTPS response m' that is accepted by r as a response to m has to be encrypted with k . The nonce k is stored by the RP in the `pendingRequests` state information. The RP only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the browser checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and r (which did not leak it either, as u did not leak it and r is honest, see (2)). The RP r cannot send responses that are encrypted by symmetric encryption keys used for outgoing HTTPS requests (all encryption keys used for encrypting responses are taken from the matching HTTPS requests and never from `pendingRequests`). This proves (4). \square

Lemma 2. For every honest relying party $r \in \text{RP}$, every $s \in \rho$, every $\langle \text{host}, \text{wkDoc} \rangle \in {}^\diamond S(r).\text{wkCache}$ it holds that $\text{wkDoc}[\text{signkey}] \equiv \text{pub}(\text{signkey}(\text{dom}^{-1}(\text{host})))$ if $\text{dom}^{-1}(\text{host})$ is an honest IdP.

Proof. First, we can see that (in an honest RP) $S(r).wkCache$ can only be populated in Line 14 (of Algorithm 9). There, the body of a received message m' is written to $S(r).wkCache$. From Line 7 we can see that m' is the response to a HTTPS message that was sent by r . Only in Lines 39ff., r can assemble (and later sent) such requests.

All such requests are sent to the path `/.well-known/spresso-info`. As the original request was stored in `pendingRequests`, in Line 14, we know that `request.host` is the domain the original request was encrypted for and finally sent to.

With the condition of this lemma we see that $\text{dom}^{-1}(\text{request.host})$ is an honest IdP, say, p . Lemma 1 applies here and we can see that p created the HTTPS response, and it was not altered by any other party. In Algorithm 10 we can see that an honest IdP responds to requests to the path `/.well-known/spresso-info` in Line 10ff. Here, p constructs a document `wkDoc` and sends this document in the body of the HTTPS response. This document is of the following form: $\langle\langle \text{signkey}, \text{pub}(s'.\text{signkey}) \rangle\rangle$. The term $s'.\text{signkey}$ is defined in Definition 47 to be $\text{signkey}(p)$ and is never changed in Algorithm 10.

Therefore, a pairing of the form $\langle \text{request.host}, x \rangle$ with $x[\text{signkey}] \equiv \text{pub}(\text{signkey}(\text{dom}^{-1}(\text{request.host})))$ is stored in $S(r).wkCache$. As this applies to all pairings in $S(r).wkCache$, this proves the lemma. \square

Definition 61. For every service token $\langle n, i \rangle$ we define a service token response for $\langle n, i \rangle$ to be an HTTPS response where the value n is contained in the body of the message. A service token request for $\langle n, i \rangle$ is an HTTPS request that triggered the service token response for $\langle n, i \rangle$.

Lemma 3. In a run ρ of $SW\mathcal{S}^{\text{auth}}$, for every state $s_j \in \rho$, every RP $r \in \text{RP}$ that is honest in s_j , every $\langle n, i \rangle \in S^j(r).serviceTokens$, the following properties hold:

1. There exists exactly one $l' < j$ such that there exists a processing step in ρ of the form

$$s_{l'} \xrightarrow[r \rightarrow \langle \langle a', f', m' \rangle \rangle]{e' \rightarrow r} s_{l'+1}$$

with e' being some events, a' and f' being addresses and m' being a service token response for $\langle n, i \rangle$.

2. There exists exactly one $l < j$ such that there exists a processing step in ρ of the form

$$s_l \xrightarrow[r \rightarrow e]{\langle a, f, m \rangle \rightarrow r} s_{l+1}$$

with e being some events, a and f being addresses and m being a service token request for $\langle n, i \rangle$.

3. The processing steps from (1) and (2) are the same, i.e., $l = l'$.
4. The service token request for $\langle n, i \rangle$, m in (2), is an HTTPS message of the following form:

$$\text{enc}_a(\langle \langle \text{HTTPReq}, n_{req}, \text{POST}, d_r, /login, x, h, b \rangle, k \rangle, \text{pub}(\text{sslkey}(d_r)))$$

for $d_r \in \text{dom}(r)$, some terms x, h, n_{req} , and a dictionary b such that

$$b[\text{eia}] \equiv \text{enc}_s(\text{sig}(\langle \text{tag}, i, S(r).FWDDomain \rangle, k_{sign}), \text{iaKey})$$

with

$$\text{tag} \equiv \text{enc}_s(\langle d_r, n_{rp} \rangle, \text{tagKey}),$$

$$i \equiv S^l(r).loginSessions[b[\text{loginSessionToken}]].\text{email},$$

$$\text{tag} \equiv S^l(r).loginSessions[b[\text{loginSessionToken}]].\text{tag},$$

$$\text{iaKey} \equiv S^l(r).loginSessions[b[\text{loginSessionToken}]].\text{iaKey}$$

for some nonces n_{rp} , and k_{sign} .

5. If the governor of i is an honest IdP, we have that $k_{sign} = \text{signkey}(\text{governor}(i))$.

Proof. (1). The service token nonce n of service tokens $\langle n, i \rangle \in {}^\diamond S^j(r).serviceTokens$ can only be contained in a response that is assembled in Lines 53ff of Algorithm 9. The n is freshly chosen in Line 67, stored (along with the identity i) to $S^j(r).serviceTokens$ (actually to $S^q(r).serviceTokens$ for some $q \leq j$) in Line 69 and sent out in the service token response in Line 70f. The service tokens stored in $S^j(r).serviceTokens$ are not used or altered anywhere else. Therefore, each service token nonce is sent in exactly one (service token) response.

(2). From Line 53 of Algorithm 9 it is easy to see that each service token response is triggered by exactly one request.

(3). Follows immediately from (2).

(4). The basic form of the encrypted HTTPS request, the host header, and the usage of the correct encryption key are enforced by Lines 26f. The *path* component is checked to be `/login` and the *method* component is checked to be `POST` in Line 53. The values of $b[eia]$, i , tag , and $iaKey$ are checked in Lines 62ff.

(5). In Line 64, the term ia is checked to be signed with the signature key stored in $S^q(r).wkCache$ indexed under the domain of the email address i (for some $q \leq j$). With Lemma 2, we can see that for the domain of the email address i this signature key is $signkey(dom^{-1}(i.domain))$. With $dom^{-1}(i.domain) = governor(i)$ we can see that ia must have been signed with the signature key of the honest IdP that governs the email address i . Further, in the same line, the contents of the signature, including the tag, are checked. □

F.2 Property A

As stated above, the Property A is defined as follows:

Definition 62. Let SWS^{auth} be an SPRESSO web system for authentication analysis. We say that SWS^{auth} is secure (with respect to Property A) if for every run ρ of SWS^{auth} , every state (S^j, E^j, N^j) in ρ , every $r \in RP$ that is honest in S^j with $S^0(r).FWDDomain$ being a domain of an FWD that is honest in S^j , every RP service token of the form $\langle n, i \rangle$ recorded in $S^j(r).serviceTokens$ and derivable from the attackers knowledge in S^j (i.e., $\langle n, i \rangle \in d_\emptyset(S^j(attack))$), it follows that the browser b owning i is fully corrupted in S^j (i.e., the value of $isCorrupted$ is `FULLCORRUPT`) or $governor(i)$ is not an honest IdP (in S^j).

We want to show that every SPRESSO web system is secure with regard to Property A and therefore assume that there exists an SPRESSO web system that is not secure. We will lead this to a contradiction and thereby show that all SPRESSO web systems are secure (with regard to Property A).

In detail, we assume: There exists an SPRESSO web system SWS^{auth} , a run ρ of SWS^{auth} , a state $s_j = (S^j, E^j, N^j)$ in ρ , a RP $r \in RP$ that is honest in S^j with $S^0(r).FWDDomain$ being a domain of an FWD that is honest in S^j , an RP service token of the form $\langle n, i \rangle$ recorded in $S^j(r).serviceTokens$ and derivable from the attackers knowledge in S^j (i.e., $\langle n, i \rangle \in d_\emptyset(S^j(attack))$), and the browser b owning i is not fully corrupted and $governor(i)$ is an honest IdP (in S^j).

We now proceed to to proof that this is a contradiction. First, we can see that for $\langle n, i \rangle$ and s_j , the conditions in Lemma 3 are fulfilled, i.e., a service token request m and a service token response m' to/from r exist, and m' is of form shown in Lemma 3 (4). Let $I := governor(i)$. We know that I is an honest IdP. As such, it never leaks its signing key (see Algorithm 10). Therefore, the signed subterm $ia := sig(\langle tag, i, S(r).FWDDomain \rangle, signkey(I))$ had to be created by the IdP I . An (honest) IdP creates signatures only in Line 25 of Algorithm 10.

Lemma 4. Under the assumption above, only the browser b can issue a request (say, m_{cert}) that triggers the IdP I to create the signed term ia . The request m_{cert} was sent by b over HTTPS using I 's public HTTPS key.

Proof. We have to consider two cases for the request m_{cert} :

(A). First, if the user is not logged in with the identity i at I (i.e., the browser b has no session cookie that carries a nonce which is a session id at I for which the identity i is marked as being logged in, compare Line 22 of Algorithm 10), then the request has to carry (in the request body) the password matching the identity i ($secretOfID(i)$). This secret is only known to b initially. Depending on the corruption status of b , we can now have two cases:

- a) If b is honest in s_j , it has not sent the secret to any party except over HTTPS to I (as defined in the definition of browsers).

- b) If b is close-corrupted, it has not sent it to any other party while it was honest (case a). When becoming close-corrupted, it discarded the secret.

I.e., the secret has been sent only to I over HTTPS or to nobody at all. The IdP I cannot send it to any other party. Therefore we know that only the browser b can send the request m_{cert} in this case.

(B). Second, if the user is logged in for the identity i at I , the browser provides a session id to I that refers to a logged in session at I . This session id can only be retrieved from I by logging in, i.e., case (A) applies, in particular, b has to provide the proper secret, which only itself and I know (see above). The session id is sent to b in the form of a cookie, which is set to secure (i.e., it is only sent back to I over HTTPS, and therefore not derivable by the attacker, see prior work) and httpOnly (i.e., it is not accessible by any scripts). The browser b sends the cookie only to I . The IdP I never sends the session id to any other party than b . The session id therefore only leaks to b and I , and never to the attacker. Hence, the browser b is the only atomic DY process which can send the request m_{cert} in this case.

We can see that in both cases, the request was sent by b using HTTPS and I 's public key: If the browser would intend to send the request without encryption, the request would not contain the password in case (A) or the cookie in case (B). The browser always uses the “correct” encryption key for any domain (as defined in $\mathcal{SM}\mathcal{S}^{\text{auth}}$). \square

As the request m_{cert} is sent over HTTPS, it cannot be altered or read by any other party. In particular, it is easy to see that at the point in the run where m_{cert} was sent, b was honest (otherwise, it would have had no knowledge of the secret anymore).

Lemma 5. *In the browser b , the request m_{cert} was triggered by `script_idp` loaded from the origin $\langle d, S \rangle$ for some $d \in \text{dom}(I)$.*

Proof. First, $\langle d, S \rangle$ for some $d \in \text{dom}(I)$ is the only origin that has access to the secret `secretOfID(i)` for the identity i (as defined in Appendix D.11).

With the general properties defined in [11] and the definition of Identity Providers in Appendix D.13, in particular their property that they only send out one script, `script_idp`, we can see that this is the only script that can trigger a request containing the secret. \square

Lemma 6. *In the browser b , the script `script_idp` receives the response to the request m_{cert} (and no other script), and at this point, the browser is still honest.*

Proof. From the definition of browser corruption, we can see that the browser b discards any information about pending requests in its state when it becomes close-corrupted, in particular any SSL keys. It can therefore not decrypt the response if it becomes close-corrupted before receiving the response.

The rest follows from the general properties defined in [11]. \square

We now know that only the script `script_idp` received the response containing the IA. For the following lemmas, we will assume that the browser b is honest. In the other case (the browser is close-corrupted), the IA ia and any information about pending HTTPS requests (in particular, any decryption keys) would be discarded from the browser's state (as seen in the proof for Lemma 6). This would be a contradiction to the assumption (which requires that the IA arrived at the RP).

Lemma 7. *After receiving ia , `script_idp` forwards the ia only to an FWD that is honest (in s_j , and therefore, also at any earlier point in the run) and a document `script_fwd` that was loaded from this FWD over HTTPS.*

Proof. We know that the browser b is either close-corrupted (in which case the ia would be discarded as it is only stored in the window structure, or, more precisely, the script states inside the window structure of the browser, which are removed when the browser becomes close-corrupted) or it is honest. In the latter case, `script_idp` (defined in Algorithm 15) opens an iframe from the `FWDDomain` that was given to it by RP. It always uses HTTPS for this request.

We can see that `script_idp` forwards the ia to the domain stored in the variable `FWDDomain` (Line 20 of Algorithm 15). This variable is set five lines earlier with the value taken from the parameters of the current document. While we cannot know the actual value of the parameter `FWDDomain` yet, we know that this parameter does not change (in the browser definition, it is only set once, when the document is loaded). We can also see that the very same parameter

was sent to I in Line 10 as the value for the FWD domain that was then signed by I in the ia . As we know the value of the FWD origin in the ia (it is $S(r).FWDDomain$), we know that the domain to which the ia is forwarded is the same.

From our assumption, we know that $S(r).FWDDomain$ is the origin of an honest FWD in s_j . It is contacted over HTTPS, so the general properties defined in [11] apply. According to the definition of forwarders (Algorithm 11), they only respond with $script_fwd$. The ia is therefore only forwarded to the FWD and its script $script_fwd$. \square

Lemma 8. *The script $script_fwd$ forwards the ia only to the script $script_rp$ loaded from the origin $\langle d_r, S \rangle$.*

Proof. The script $script_idp$ that runs in the honest browser b forwards the (then encrypted) IA along with the tag to $script_fwd$. From the definition of the IdP script (Algorithm 15) it is clear that the tag that is forwarded along with the encrypted IA is the same that was signed by the IdP.

This script (Algorithm 11) tries to decrypt the tag (once it receives a matching key) and sends a `postMessage` containing the encrypted IA to the domain contained in the tag, which is d_r .

The protocol part of the origin is HTTPS. The only document that r delivers and which receives `postMessages` is $script_rp$, and this therefore is the only script that can receive this `postMessage`. \square

Lemma 9. *From the RP document, the EIA is only sent to the RP r and over HTTPS.*

Proof. This follows immediately from the definition of $script_rp$ (see Algorithm 13, in particular Line 36 in conjunction with Line 3) and the fact that the RP document must have been loaded from the origin $\langle d_r, S \rangle$ (as shown above).

With Lemmas 6–9 we see that the ia , once it was signed by I , was transferred only to r , the browser b , and to an honest forwarder. It cannot be known to the attacker or any corrupted party, as none of the listed parties leak it to any corrupted party or the attacker.

Now, for $\langle n, i \rangle$ to be created and recorded in $S^j(r)$, a message m as shown above has to be created and sent. This can only be done with knowledge of eia . From their definitions, we can see that neither I , r nor any forwarder create such a message, with the only option left being b . If b sends such a request, it is the only party able to read the response (see general security properties in [11]) and it will not do anything with the contents of the response (see Algorithm 13), in particular not leak it to the attacker or any corrupted party.

This is a contradiction to the assumption, where we assumed that $\langle n, i \rangle \in d_0(S^j(\text{attacker}))$. This shows every SWS^{auth} is secure in the sense of Property A. ■

F.3 Property B

As stated above, Property B is defined as follows:

Definition 63. *Let SWS^{auth} be an SPRESSO web system. We say that SWS^{auth} is secure (with respect to Property B) if for every run ρ of SWS^{auth} , every state (S^j, E^j, N^j) in ρ , every $r \in RP$ that is honest in S^j with $S^0(r).FWDDomain$ being a domain of an FWD that is honest in S^j , every RP service token of the form $\langle n, i \rangle$ recorded in $S^j(r).serviceTokens$, with the request corresponding to $\langle n, i \rangle$ sent by some $b \in B$ which is honest in S^j , b owns i .*

Applying Lemma 3 (1–4), we call the request corresponding to $\langle n, i \rangle$ (or service token request) m and its response m' , and (as in Lemma 3 (2)) we refer to the state of SWS^{auth} in the run ρ where r processes m by s_l .

Lemma 10. *The request m was sent by $script_rp$ loaded from the origin $\langle d_r, S \rangle$ where d_r is some domain of r .*

Proof. The request m is XSRF protected. In Algorithm 9, Line 54, RP checks the presence of the Origin header and its value. If the request m was initiated by a document from a different origin than $\langle d_r, S \rangle$, the (honest!) browser b would have added an Origin header that would not pass this test (or no Origin header at all), according to the browser definition. The script $script_rp$ is the only script that the honest party r sends as a response and that sends a request to r . \square

Lemma 11. *The request m contains a nonce $loginSessionToken$ such that*

$$S^l(r).loginSessions[loginSessionToken].email \equiv i'$$

and b owns i' , i.e., $ownerOfID(i') = b$.

Proof. With Lemma 10 we know that the request was sent by $script_rp$. In Algorithm 13 defining $script_rp$, in Line 35, the body of the request m is assembled (and this is the only line where this script sends a request that contains the same path as m). The login session token is taken from the script's state ($loginSessionToken$). This part of the state is initially set to \perp and is only changed in Line 14. There, it is taken from the response to the start login XHR issued in Line 8 (the request and response are coupled using $refXHR$ which is tracked in the script's state). In Line 6, the script selects one of the browser's identities (which are the identities that the browser owns, by the definition of browsers in Appendix D.11). This identity is then used in the start login XHR.

When receiving this request (which is an HTTPS message, and therefore, cannot be altered nor read by the attacker), ultimately, the function $SENDSTARTLOGINRESPONSE$ (Algorithm 8) is called. There are two cases how this function can be called (see Line 36 of Algorithm 9):

- If the well-know cache of r already contains an entry for the host contained in the email address, $SENDSTARTLOGINRESPONSE$ is called immediately with the email address contained in the request's body.
- Else, the email address in the request's body is stored, together with the request's HTTP nonce, the HTTPS encryption key and other data, in the subterm $pendingDNS$ of r 's state. From there, it is later moved to $pendingRequests$ (Line 21). Finally, in Line 15, $SENDSTARTLOGINRESPONSE$ is called.

We will come back to these two cases further down.

After $SENDSTARTLOGINRESPONSE$ is called, a new $loginSessionToken$ is chosen and in the dictionary $S^x(r).loginSessions[loginSessionToken]$ the email address (along with other data) is stored (for some x).

The $loginSessionToken$ is then sent as a response to m , in particular, it is encrypted with the symmetric key k contained in the request. In the first case listed above, the k is immediately retrieved from the request. Otherwise, the relationship between k and the email address is preserved in any case: If the receiver can decrypt the response to m , it sent the email address i' in the request.

As explained above, $script_rp$ takes the $loginSessionToken$ from the response body and stores it in its state to later use it in the request m . Therefore the start login XHR described above must have taken place before m , i.e., $x < l$.

The entries in the dictionary $loginSessions$ can not be altered and only be removed when a service token request with the corresponding value of $loginSessionToken$ is processed. As each $loginSessionToken$ is not leaked to any other party except r , we know that $S^l(r).loginSessions[loginSessionToken].email \equiv i'$. As shown above, due to the way i' is selected by the script, b owns i' . \square

With Lemma 11, we can now show that $i = i'$: In Line 68 of Algorithm 9, the service token is assembled. In particular, i is chosen to be $S^l(r).loginSessions[loginSessionToken].email$, and therefore $i = i'$ and b owns i . \blacksquare

G Indistinguishability of Web Systems

Definition 64 (Web System Command and Schedule). *We call a term ζ a web system command (or simply, command) if ζ is of the form*

$$\langle i, j, \tau_{process}, cmd_{switch}, cmd_{window}, \tau_{script}, url \rangle$$

The components are defined as follows:

- $i \in \mathbb{N}$,
- $j \in \mathbb{N}$,
- $cmd_{switch} \in \{1, 2, 3\}$,
- $cmd_{window} \in \mathbb{N}$,

- $\tau_{script} \in \mathcal{T}_0(V_{script} \cup \{x\})$ with x being a variable and V_{script} the set of placeholders for scripting processes (see Definition 18).
- $\tau_{process} \in \mathcal{T}_0(V_{process} \cup \{x\})$ with x being a variable and $V_{process}$ the set of placeholders (see Definition 4).
- $url \in \text{URLs}$ with URLs being the set of all valid URLs (see Definition 25).

We call a (finite) sequence $\sigma = \langle \zeta_1, \dots, \zeta_n \rangle$, with ζ_1, \dots, ζ_n being web system commands, a web system schedule (or simply, schedule).

Definition 65 (Induced Processing Step). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ be a web system and

$$(S, E, N) \xrightarrow[p \rightarrow E_{out}]{\langle a, f, m \rangle \rightarrow p} (S', E', N')$$

be a processing step of \mathcal{W} (as in Definition 13) with $E = (e_1, e_2, \dots)$ and

$$\zeta = \langle i, j, \tau_{process}, \text{cmd}_{switch}, \text{cmd}_{window}, \tau_{script}, url \rangle$$

a web system command. We say that this processing step is induced by ζ iff

1. $e_i = \langle a, f, m \rangle$.
2. Under a lexicographic ordering of \mathcal{W} , p is the j -th process in \mathcal{W} with $a \in I^p$.
3. $E' = E_{out} \cdot (e_1, \dots, e_{i-1}, e_{i+1}, \dots)$.
4. If p is a (web) attacker process or p is a corrupted browser (i.e., $S(p).\text{isCorrupted} \neq \perp$), then $E_{out} = \langle e_{out} \rangle$ with $\langle S'(p), e_{out} \rangle = \tau_{process}[\langle e_i, s \rangle / x] \downarrow$.
5. If p is an honest browser (i.e., $S(p).\text{isCorrupted} \equiv \perp$) and $m \equiv \text{TRIGGER}$, the browser relation behaves as follows and E_{out} and $S'(p)$ are obtained accordingly:
 - (a) If $\text{cmd}_{switch} = 1$, the browser relation chooses $\text{switch} = 1$ in Line 9 of Algorithm 7 and \bar{w} in Line 11 of Algorithm 7 such that \bar{w} is the cmd_{window} -th window in the tree of browser's state $S(p).\text{windows}$. If this script is not the attacker script, the browser (deterministically) executes the script in this window. Otherwise, in Line 10 of Algorithm 5, the browser relation chooses the output of the script (of this window) as $out^\lambda = \tau_{script}[\text{in}/x] \downarrow$ with the variable in (deterministically) chosen in Line 9 of Algorithm 5.
 - (b) If $\text{cmd}_{switch} = 2$, the browser relation chooses $\text{switch} = 2$ in Line 9 of Algorithm 7 and protocol, host, domain, path, parameters in Line 17ff. of Algorithm 7 such that $url = \langle \text{URL}, \text{protocol}, \text{host}, \text{path}, \text{parameters} \rangle$.
 - (c) If $\text{cmd}_{switch} = 3$, the browser relation chooses $\text{switch} = 3$ in Line 9 of Algorithm 7 and \bar{w} in Line 24 of Algorithm 7 such that \bar{w} is the cmd_{window} -th window in the tree of browser's state $S(p).\text{windows}$. (The browser then starts to reload the document in this window.)

We write

$$(S, E, N) \xrightarrow{\zeta} (S', E', N').$$

Corollary 1. In some cases a command $\sigma = \langle i, j, \tau_{process}, \text{cmd}_{switch}, \text{cmd}_{window}, \tau_{script}, url \rangle$ does not induce a processing step under the configuration (S, E, N) in a web system: If $i > |E|$, a processing step cannot be induced. The same applies if j does not refer to an existing process. Also, if the command schedules a TRIGGER message to be delivered to a browser p , $\text{cmd}_{switch} \in \{1, 3\}$, and $\text{cmd}_{window} > |\text{Subwindows}(S(p))|$ (i.e., the command chooses a window of the browser p , which does not exist), then no processing step can be induced.

Definition 66 (Induced Run). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ be a web system, $\sigma = \langle \zeta_1, \dots, \zeta_n \rangle$ be a finite web system schedule, and N^0 be an infinite sequence of pairwise disjoint nonces. We say that a finite run $\rho = ((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ of the system \mathcal{W} is induced by σ under nonces N^0 iff for all $1 \leq i \leq n$, ζ_i induces the processing step

$$(S^{i-1}, E^{i-1}, N^{i-1}) \xrightarrow{\zeta_i} (S^i, E^i, N^i).$$

We denote the set of runs induced by σ under all infinite sequences of pairwise disjoint nonces N^0 by $\sigma(\mathcal{WS})$.

To define the notion of indistinguishability for web systems, we need to define the notion of static equivalence of terms in our model. This definition follows the notion of static equivalence by Abadi and Fournet [1].

Definition 67 (Static Equivalence). Let $t_1, t_2 \in \mathcal{T}_{\mathcal{N}}(V)$ be two terms with V a set of variables. We say that t_1 and t_2 are statically equivalent, written $t_1 \approx t_2$, iff for all terms $M, N \in \mathcal{T}_0(\{x\})$ with x a variable and $x \notin V$, it holds true that

$$M[t_1/x] \equiv N[t_1/x] \quad \Leftrightarrow \quad M[t_2/x] \equiv N[t_2/x].$$

Definition 68 (Web System with Distinguished Attacker). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ be a web system with \mathcal{W} partitioned into Hon, Web, and Net (as in Definition 21). Let $\text{attacker} \in \mathcal{W}$ be an attacker process (out of $\text{Web} \cup \text{Net}$). We call $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0, \text{attacker})$ a web system with the distinguished attacker attacker .

Definition 69 (Indistinguishability). Let $\mathcal{WS}_0 = (\mathcal{W}_0, \mathcal{S}_0, \text{script}_0, E_0^0, p_0)$, $\mathcal{WS}_1 = (\mathcal{W}_1, \mathcal{S}_1, \text{script}_1, E_1^0, p_1)$ be web systems with a distinguished attacker. We call \mathcal{WS}_0 and \mathcal{WS}_1 indistinguishable under the schedule σ iff for every schedule σ and every $i \in \{0, 1\}$, we have that for every run $\rho \in \sigma(\mathcal{WS}_i)$ there exists a run $\rho' \in \sigma(\mathcal{WS}_{1-i})$ such that $\rho(p_i) \approx \rho'(p_{1-i})$.

We call \mathcal{WS}_0 and \mathcal{WS}_1 indistinguishable iff they are indistinguishable under all schedules σ .

H Formal Proof of Privacy

We will here first describe the precise model that we use for privacy. After that, we define an equivalence relation between configurations, which we will then use in the proof of privacy.

H.1 Formal Model of SPRESSO for Privacy Analysis

Definition 70 (Challenge Browser). Let dr some domain and $b(dr)$ a DY process. We call $b(dr)$ a challenge browser iff b is defined exactly the same as a browser (as described in Appendix C) with two exceptions: (1) the state contains one more property, namely challenge, which initially contains the term \top . (2) Algorithm 4 is extended by the following at its very beginning: It is checked if a message m is addressed to the domain CHALLENGE (which we call the challenger domain). If m is addressed to this domain and no other message m' was addressed to this domain before (i.e., $\text{challenge} \neq \perp$), then m is changed to be addressed to the domain dr and challenge is set to \perp to recorded that a message was addressed to CHALLENGE.

Definition 71 (Deterministic DY Process). We call a DY process $p = (I^p, Z^p, R^p, s_0^p)$ deterministic iff the relation R^p is a (partial) function.

We call a script R_{script} deterministic iff the relation R_{script} is a (partial) function.

Definition 72 (SPRESSO Web System for Privacy Analysis). Let $\mathcal{SW}\mathcal{S} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ be an SPRESSO web system with $\mathcal{W} = \text{Hon} \cup \text{Web} \cup \text{Net}$, $\text{Hon} = \text{B} \cup \text{RP} \cup \text{IDP} \cup \text{FWD} \cup \text{DNS}$ (as described in Appendix D.1), $\text{RP} = \{r_1, r_2\}$, $\text{FWD} = \{\text{fwd}\}$, $\text{DNS} = \{\text{dns}\}$, r_1 and r_2 two (honest) relying parties, fwd an honest forwarder, dns an honest DNS server. Let $\text{attacker} \in \text{Web}$ be some web attacker. Let dr be a domain of r_1 or r_2 and $b(dr)$ a challenge browser. Let $\text{Hon}' := \{b(dr)\} \cup \text{RP} \cup \text{FWD} \cup \text{DNS}$, $\text{Web}' := \text{Web}$, and $\text{Net}' := \emptyset$ (i.e., there is no network attacker). Let $\mathcal{W}' := \text{Hon}' \cup \text{Web}' \cup \text{Net}'$. Let $\mathcal{S}' := \mathcal{S} \setminus \{\text{script}_{\text{idp}}\}$ and script' be accordingly. We call $\mathcal{SW}\mathcal{S}^{\text{priv}}(dr) = (\mathcal{W}', \mathcal{S}', \text{script}', E^0, \text{attacker})$ an SPRESSO web system for privacy analysis iff the domain fwd domain is the only domain assigned to fwd , the domain dr_1 the only domain assigned to r_1 , and dr_2 the only domain assigned to r_2 . Both, r_1 and r_2 are configured to use the forwarder fwd , i.e., in their state FWD domain is set to fwd domain. The browser $b(dr)$ owns exactly one email address and this email address is governed by some attacker. All honest parties (in Hon) are not corruptible, i.e., they ignore any CORRUPT message. Identity providers are assumed to be dishonest, and hence, are subsumed by the web attackers (which govern all identities). In the initial state s_0^b of the (only) browser in \mathcal{W}' and in the initial states $s_0^{r_1}, s_0^{r_2}$ of both relying parties, the DNS address is $\text{addr}(\text{dns})$. Further, wkCache in the initial states $s_0^{r_1}, s_0^{r_2}$ is equal and contains a public key for each domain registered in the DNS server (i.e., the relying parties already know some public key to verify SPRESSO identity assertions from all domains known in the system and they do not have to fetch them from IdP).

As all parties in an SPRESSO web system for privacy analysis are either web attackers, browsers, or deterministic processes and all scripting processes are either the attacker script or deterministic, it is easy to see that in SPRESSO web systems for privacy analysis with configuration (S, E, N) a command ζ induces at most one processing step. We further note that, under a given infinite sequence of nonces N^0 , all schedules σ induce at most one run $\rho = ((S^0, E^0, N^0), \dots, (S^i, E^i, N^i), \dots, (S^{|\sigma|}, E^{|\sigma|}, N^{|\sigma|}))$ as all of its commands induce at most one processing step for the i -th configuration.

We will now define our privacy property for SPRESSO:

Definition 73 (IdP-Privacy). *Let*

$$\begin{aligned} \mathcal{SWS}_1^{priv} &:= \mathcal{SWS}^{priv}(dr_1) = (\mathcal{W}_1, \mathcal{S}, \text{script}, E^0, \text{attacker}_1) \text{ and} \\ \mathcal{SWS}_2^{priv} &:= \mathcal{SWS}^{priv}(dr_2) = (\mathcal{W}_2, \mathcal{S}, \text{script}, E^0, \text{attacker}_2) \end{aligned}$$

be SPRESSO web systems for privacy analysis. Further, we require $\text{attacker}_1 = \text{attacker}_2 =: \text{attacker}$ and for $b_1 := b(dr_1)$, $b_2 := b(dr_2)$ we require $S(b_1) = S(b_2)$ and $\mathcal{W}_1 \setminus \{b_1\} = \mathcal{W}_2 \setminus \{b_2\}$ (i.e., the web systems are the same up to the parameter of the challenge browsers). We say that \mathcal{SWS}^{priv} is IdP-private iff \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} are indistinguishable.

H.2 Definition of Equivalent Configurations

Let $\mathcal{SWS}_1^{priv} = (\mathcal{W}_1, \mathcal{S}, \text{script}, E^0, \text{attacker})$ and $\mathcal{SWS}_2^{priv} = (\mathcal{W}_2, \mathcal{S}, \text{script}, E^0, \text{attacker})$ be SPRESSO web systems for privacy analysis. Let (S_1, E_1, N_1) be a configuration of \mathcal{SWS}_1^{priv} and (S_2, E_2, N_2) be a configuration of \mathcal{SWS}_2^{priv} .

Definition 74 (Proto-Tags). *We call a term of the form $\text{enc}_s(\langle y, n \rangle, k)$ with the variable y as a placeholder for a domain, and n and k some nonces a proto-tag.*

Definition 75 (Term Equivalence up to Proto-Tags). *Let $\theta = \{a_1, \dots, a_l\}$ be a finite set of proto-tags. Let t and t' be terms. We call t_1 and t_2 term-equivalent under a set of proto-tags θ iff there exists a term $\tau \in \mathcal{T}_{\mathcal{N}}(\{x_1, \dots, x_l\})$ such that $t_1 = (\tau[a_1/x_1, \dots, a_l/x_l])[dr_1/y]$ and $t_2 = (\tau[a_1/x_1, \dots, a_l/x_l])[dr_2/y]$. We write $t_1 \rightleftharpoons_{\theta} t_2$.*

We say that two finite sets of terms D and D' are term-equivalent under a set of proto-tags θ iff $|D| = |D'|$ and, given a lexicographic ordering of the elements in D of the form $(d_1, \dots, d_{|D|})$ and the elements in D' of the form $(d'_1, \dots, d'_{|D'|})$, we have that for all $i \in \{1, \dots, |D|\}$: $d_i \rightleftharpoons_{\theta} d'_i$. We then write $D \rightleftharpoons_{\theta} D'$.

Definition 76 (Equivalence of HTTP Requests). *Let m_1 and m_2 be (potentially encrypted) HTTP requests and $\theta = \{a_1, \dots, a_l\}$ be a finite set of proto-tags. We call m_1 and m_2 δ -equivalent under a set of proto-tags θ iff $m_1 \rightleftharpoons_{\theta} m_2$ or all subterms are equal with the following exceptions:*

1. the Host value and the Origin/Referer headers in both requests are the same except that the domain dr_1 in m_1 can be replaced by dr_2 in m_2 ,
2. the HTTP body g_1 of m_1 and the HTTP body g_2 of m_2 are (I) term-equivalent under θ , (II) for $j \in \{1, 2\}$ if $g_j[\text{eia}] \sim \text{enc}_s(\text{sig}(\langle \text{enc}_s(\langle dr_j, * \rangle, *), *, *, \text{fwddomain} \rangle, *), *)$ and the origin (HTTP header) of HTTP message in m_j is $\langle dr_j, \mathcal{S} \rangle$ then the receiver of this message is r_j , and (III) if g_1 contains a dictionary key `loginSessionToken` then there exists an $l' \in L$ such that $g_1[\text{loginSessionToken}] \equiv l'$, and
3. if m_1 is an encrypted HTTP request then and only then m_2 is an encrypted HTTP request and the keys used to encrypt the requests have to be the correct keys for dr_1 and dr_2 respectively.

We write $m_1 \simeq_{\theta} m_2$.

Definition 77 (Extracting Entries from Login Sessions). *Let t_1, t_2 be dictionaries over \mathcal{N} and $\mathcal{T}_{\mathcal{N}}$, θ be a finite set of proto-tags, and d a domain. We call t_1 and t_2 η -equivalent iff t_2 can be constructed from t_1 as follows: For every proto-tag $a \in \theta$, we remove the entry identified by the dictionary key i for which it holds that $\pi_4(t_1[i]) \equiv a[d/y]$, if any. We denote the set of removed entries by D . We write $t_1 \succeq_d^{\theta}(t_2, D)$.*

Definition 78. Let a be a proto-tag, S_1 and S_2 be states of SPRESSO web systems for privacy analysis, and l a nonce. We call l a login session token for the proto-tag a , written $l \in \text{loginSessionTokens}(a, S_1, S_2)$ iff for any $i \in \{1, 2\}$ and any $j \in \{1, 2\}$ we have that $\pi_4(S_i(r_j).\text{loginSessions}[l]) = a[dr_j/y]$.

Definition 79 (Equivalence of States). Let θ be a set of proto-tags and H be a set of nonces. Let $K := \{k \mid \exists n : \text{enc}_s(\langle y, n \rangle, k) \in \theta\}$. We call S_1 and S_2 γ -equivalent under (θ, H) iff the following conditions are met:

1. $S_1(\text{fwd}) = S_2(\text{fwd})$, and
 2. $S_1(\text{dns}) = S_2(\text{dns})$, and
 3. $S_1(r_1)$ equals $S_2(r_1)$ except for the subterms `pendingDNS`, `loginSessions` and `serviceTokens`, and
 4. $S_1(r_2)$ equals $S_2(r_2)$ except for the subterms `pendingDNS`, `loginSessions` and `serviceTokens`, and
 5. for two sets of terms D and D' : $S_1(r_1).\text{loginSessions} \succeq_{dr_1}^\theta (S_2(r_1).\text{loginSessions}, D)$, $S_2(r_2).\text{loginSessions} \succeq_{dr_2}^\theta (S_1(r_2).\text{loginSessions}, D')$, and $D \equiv_\theta D'$, and
 6. for all entries x in the subterms `pendingDNS` of $S_1(r_1)$, $S_1(r_1)$, $S_1(r_1)$, and $S_1(r_1)$ it holds true that $\pi_2 x.\text{host}$ is not a domain name known to the DNS server; and
 7. the subterms `pendingRequest` of $S_1(r_1)$, $S_1(r_2)$, $S_2(r_1)$, and $S_2(r_2)$ are $\langle \rangle$, and
 8. the subterm `wkCache` of $S_1(r_1)$, $S_1(r_2)$, $S_2(r_1)$, and $S_2(r_2)$ are equal and contain a public key for each domain registered in the DNS server; and
 9. $\forall k \in K: k \notin d_\theta(\bigcup_{i \in \{1, 2\}, A \in \text{Web} \cup \text{Net} \cup \{\text{dns}, \text{fwd}\}} S_i(A))$
 10. for each attacker A : $S_1(A) \equiv_\theta S_2(A)$, and
 11. for all $a \in \theta$ and all attackers A we have that $\nexists l \in \text{loginSessionTokens}(a, S_1, S_2)$ such that l is a subterm of $S_1(A)$ or $S_2(A)$.
 12. $S_1(b_1)$ equals $S_2(b_2)$ except for for the subterms `challenge`, `pendingDNS`, `pendingRequests`, `windows` and we have that
 - (a) $S_1(b_1).\text{challenge} = dr_1 \wedge S_2(b_2).\text{challenge} = dr_2$ or $S_1(b_1).\text{challenge} = S_2(b_2).\text{challenge} = \perp$, and
 - (b) $|S_1(b_1).\text{pendingDNS}| = |S_2(b_2).\text{pendingDNS}| =: j$, for all $i \in \{1, \dots, j\}$, $q_1 := \pi_i(S_1(b_1).\text{pendingDNS})$, $q_2 := \pi_i(S_2(b_2).\text{pendingDNS})$ we have that $\pi_1(q_1) = \pi_1(q_2) \in \mathcal{N}$ and for $v_1 := \pi_2(q_1)$ and $v_2 := \pi_2(q_2)$:
 - i. $\pi_1(v_1) = \pi_1(v_2)$, and
 - ii. $\pi_3(v_1) = \pi_3(v_2)$, and
 - iii. $\pi_1(v_1)$ is either a nonce ($\in \mathcal{N}$) or a term of the form $\langle x, y \rangle$ with $x \in \mathcal{N}$ a nonce and $y \in \mathcal{N} \cup \{\perp\}$ a nonce or \perp , and
 - iv. if $\pi_2(v_1).\text{host} = dr_1 \wedge \pi_2(v_2).\text{host} = dr_2$, then $\pi_2(v_1) \simeq_\theta \pi_2(v_2) \wedge \pi_2(v_1).\text{nonce} \in H$, else $\pi_2(v_1) \equiv_\theta \pi_2(v_2) \wedge \pi_2(v_1).\text{nonce} \notin H \wedge \nexists l \in L$ such that l is a subterm of $\pi_2(v_1)$,
- and
- (c) $|S_1(b_1).\text{pendingRequests}| = |S_2(b_2).\text{pendingRequests}| =: j$, for all $i \in \{1, \dots, j\}$, $v_1 := \pi_i(S_1(b_1).\text{pendingRequests})$, $v_2 := \pi_i(S_2(b_2).\text{pendingRequests})$ we have that
 - i. $\pi_1(v_1) = \pi_1(v_2)$, and
 - ii. $\pi_3(v_1) = \pi_3(v_2)$, and
 - iii. $\pi_1(v_1)$ is either a nonce ($\in \mathcal{N}$) or a term of the form $\langle x, y \rangle$ with $x \in \mathcal{N}$ a nonce and $y \in \mathcal{N} \cup \{\perp\}$ a nonce or \perp , and
 - iv. if $\pi_2(v_1).\text{host} = dr_1 \wedge \pi_2(v_2).\text{host} = dr_2$, then $\pi_2(v_1) \simeq_\theta \pi_2(v_2) \wedge \pi_2(v_1).\text{nonce} \in H \wedge \pi_4(v_1) \in \text{addr}(r_1) \wedge \pi_4(v_2) \in \text{addr}(r_2)$, else $\pi_2(v_1) \equiv_\theta \pi_2(v_2) \wedge \pi_2(v_1).\text{nonce} \notin H \wedge \pi_4(v_1) = \pi_4(v_2) \wedge \nexists l \in L$ such that l is a subterm of $\pi_2(v_1)$,
- and
- (d) there is no $k \in K$ such that

$$k \in d_{\mathcal{N} \setminus \{k\}}(\{S_1(b_1).\text{pendingRequests}, S_2(b_2).\text{pendingRequests}, S_1(b_1).\text{pendingDNS}, S_2(b_2).\text{pendingDNS}\})$$

(i.e., k cannot be derived from these terms by any party unless it knows k), and

- (e) $S_1(b_1).windows$ equals $S_2(b_2).windows$ with the exception of the subterms `location`, `referrer`, `scriptstate`, and `scriptinputs` of some document terms pointed to by $\text{Docs}^+(S_1(b_1)) = \text{Docs}^+(S_2(b_2)) =: J$. For all $j \in J$ we have that:
- i. there is no $k \in K$ such that

$$k \in d_{\mathcal{N} \setminus \{k\}}(\{S_1(b_1).j.location, S_2(b_2).j.location, \\ S_1(b_1).j.referrer, S_2(b_2).j.referrer\})$$

- ii. if $S_1(b_1).j.origin \in \{\langle dr_1, S \rangle, \langle dr_2, S \rangle\}$ then $S_1(b_1).j.script \in \{\text{script_rp}, \text{script_rp_redir}\}$, and
- iii. if $S_1(b_1).j.origin \equiv \langle \text{fwddomain}, S \rangle$ then $S_1(b_1).j.script \equiv \text{script_fwd}$, and
- iv. if $S_1(b_1).j.origin \in \{\langle dr_1, S \rangle, \langle dr_2, S \rangle\}$ and $S_1(b_1).j.script \equiv \text{script_rp}$ then
 - A. $S_1(b_1).j.location$ and $S_2(b_2).j.location$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2 , and
 - B. $S_1(b_1).j.referrer$ and $S_2(b_2).j.referrer$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2 , and
 - C. $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$ and if $\exists l \in L$ such that l is a subterm of $S_1(b_1).j.scriptstate$, then $S_1(b_1).j.location.host \equiv dr_1$ and $S_2(b_2).j.location.host \equiv dr_2$, and
 - D. for $p \in \{$

$$\begin{aligned} &\langle \text{XMLHTTPREQUEST}, *, * \rangle, \\ &\langle \text{POSTMESSAGE}, *, \langle \text{fwddomain}, S \rangle, \text{ready} \rangle, \\ &\langle \text{POSTMESSAGE}, *, \langle \text{fwddomain}, S \rangle, \langle \text{eia}, * \rangle \end{aligned}$$

$\}$ we have $S_1(b_1).j.scriptinputs|p \equiv_{\theta} S_2(b_2).j.scriptinputs|p$, and

- E. if $\exists l \in L$ such that l is a subterm of $S_1(b_1).j.scriptinputs$, then $S_1(b_1).j.location.host \equiv dr_1$ and $S_2(b_2).j.location.host \equiv dr_2$, and
- F. $\forall k \in K$: k is not contained in any subterm of $S_1(b_1).j.scriptstate$ except for $S_1(b_1).j.scriptstate.tagKey$, and
 - $S_1(b_1).j.origin \neq \langle dr_1, S \rangle$
 $\implies k \neq S_1(b_1).j.scriptstate.tagKey$, and
 - $S_1(b_1).j.origin \neq \langle dr_1, S \rangle$
 $\implies k \notin d_{\theta}(S_1(b_1).j.scriptinputs)$, and
 - $S_2(b_2).j.origin \neq \langle dr_2, S \rangle$
 $\implies k \neq S_2(b_2).j.scriptstate.tagKey$, and
 - $S_2(b_2).j.origin \neq \langle dr_2, S \rangle$
 $\implies k \notin d_{\theta}(S_2(b_2).j.scriptinputs)$, and
- v. if $S_1(b_1).j.origin \in \{\langle dr_1, S \rangle, \langle dr_2, S \rangle\}$ and $S_1(b_1).j.script \neq \text{script_rp}$ ²² then
 - A. $S_1(b_1).j.location$ and $S_2(b_2).j.location$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2 , and
 - B. $S_1(b_1).j.referrer$ and $S_2(b_2).j.referrer$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2 , and
 - C. $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$ and if $\exists l \in L$ such that l is a subterm of $S_1(b_1).j.scriptstate$, then $S_1(b_1).j.location.host \equiv dr_1$ and $S_2(b_2).j.location.host \equiv dr_2$, and
 - D. there is no $k \in K$ such that $k \in d_{\mathcal{N} \setminus \{k\}}(\{S_1(b_1).j.scriptstate\})$
- vi. if $S_1(b_1).j.origin = \langle \text{fwddomain}, S \rangle$ then
 - A. $S_1(b_1).j.location \equiv_{\theta} S_2(b_2).j.location$, and
 - B. $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$, and
 - C. for $p = \langle \text{POSTMESSAGE}, *, *, \langle \text{tagKey}, * \rangle \rangle$ and $x_1 = S_1(b_1).j.scriptinputs|p$ and $x_2 = S_2(b_2).j.scriptinputs|p$ we have that for all $i \in \{1, \dots, |x|\}$:

²²It immediately follows that $S_1(b_1).j.script \equiv \text{script_rp_redir}$ in this case.

- $\pi_2(\pi_i(x_1)) \equiv_{\theta} \pi_2(\pi_i(x_2))$, and
 - $\pi_1(\pi_3(\pi_i(x_1))) \equiv_{\theta} \pi_1(\pi_3(\pi_i(x_2)))$ or
 $\pi_1(\pi_3(\pi_i(x_1))) = dr_1 \wedge \pi_1(\pi_3(\pi_i(x_2))) = dr_2$, and
 - $\pi_2(\pi_3(\pi_i(x_1))) \equiv_{\theta} \pi_2(\pi_3(\pi_i(x_2)))$, and
 - $\pi_4(\pi_i(x_1)) \equiv_{\theta} \pi_4(\pi_i(x_2))$, and
- vii. if $S_1(b_1).j.origin \notin \{\langle dr_1, S \rangle, \langle dr_2, S \rangle, \langle fwdomain, S \rangle\}$ then
- A. $S_1(b_1).j.location \equiv_{\theta} S_2(b_2).j.location$, and
 - B. $S_1(b_1).j.referrer \equiv_{\theta} S_2(b_2).j.referrer$, and
 - C. $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$, and
 - D. $S_1(b_1).j.scriptinputs \equiv_{\theta} S_2(b_2).j.scriptinputs$, and
 - E. there is no $k \in K$ such that $k \in d_{\mathcal{N} \setminus \{k\}}(\{S_1(b_1).j.scriptstate, S_1(b_1).j.scriptinputs\})$, and
 - F. $\nexists l \in L$ such that l is a subterm of $S_1(b_1).j.scriptstate$ or of $S_1(b_1).j.scriptinputs$, and
- (f) for $x \in \{\text{cookies}, \text{localStorage}, \text{sessionStorage}, \text{sts}\}$ we have that $S_1(b_1).x \equiv_{\theta} S_2(b_2).x$. For the domains dr_1 and dr_2 there are no entries in the subterms x .

Definition 80 (Equivalence of Events). Let θ be a set of proto-tags, L be a set of login session tokens, H be a set of nonces, and $K := \{k \mid \exists n : \text{enc}_s(\langle y, n \rangle, k) \in \theta\}$. We call $E_1 = (e_1^{(1)}, e_2^{(1)}, \dots)$ and $E_2 = (e_1^{(2)}, e_2^{(2)}, \dots)$ β -equivalent under (θ, L, H) iff all of the following conditions are satisfied for every $i \in \mathbb{N}$:

1. One of the following conditions holds true:
 - (a) $e_i^{(1)} \equiv_{\theta} e_i^{(2)}$ and if $e_i^{(1)}$ contains an HTTP(S) message (i.e., HTTP(S) request or HTTP(S) response), then the HTTP nonce of this HTTP(S) message is not contained in H , or
 - (b) $e_i^{(1)}$ is a DNS request from b_1 to dns for dr_1 and $e_i^{(2)}$ is a DNS request from b_2 to dns for dr_2 , or
 - (c) $e_i^{(1)}$ and $e_i^{(2)}$ are both DNS requests from any party except dns addressed to dns for a domain unknown to the DNS server, or
 - (d) $e_i^{(1)}$ is a DNS response from dns to b_1 for a DNS request for dr_1 and $e_i^{(2)}$ is a DNS response from dns to b_2 for a DNS request for dr_2 , or
 - (e) $e_i^{(1)}$ is an HTTP request m_1 from b_1 to r_1 and $e_i^{(2)}$ is an HTTP request m_2 from b_2 to r_2 , $m_1 \simeq_{\theta} m_2$, and both requests are unencrypted or encrypted (i.e., m_1 and m_2 are the content of the encryption) and $m_1.nonce \in H$, or
 - (f) $e_i^{(1)}$ is an HTTP(S) response from r_1 to b_1 and $e_i^{(2)}$ is an HTTP(S) response from r_2 to b_2 , and their HTTP messages m_1 (contained in $e_i^{(1)}$) and m_2 (contained in $e_i^{(2)}$) are the same except for the HTTP body $g_1 := m_1.body$ and the HTTP body $g_2 := m_2.body$ which have to be $g_1 \equiv_{\theta} g_2$ and $m_1.nonce \in H$ and if g_1 contains a dictionary key `loginSessionToken` then there exists an $l' \in L$ such that $g_1[\text{loginSessionToken}] \equiv l'$.
2. If there exists $l \in L$ such that l is a subterm of $e_i^{(1)}$ or $e_i^{(2)}$ then we have that $e_i^{(1)}$ is a message from b_1 to r_1 and $e_i^{(2)}$ is a message from b_2 to r_2 or we have that $e_i^{(1)}$ is a message from r_1 to b_1 and $e_i^{(2)}$ is a message from r_2 to b_2 .
3. If there exists $k \in K$ such that $k \in d_{\mathcal{N} \setminus \{k\}}(\{e_i^{(1)}, e_i^{(2)}\})$ then $e_i^{(1)}$ is an HTTP(S) response from r_1 to b_1 and $e_i^{(2)}$ is an HTTP(S) response from r_2 to b_2 and the bodies of both HTTP messages are of the form $\langle \langle \text{tagKey}, k \rangle, *, * \rangle$.
4. If $e_i^{(1)}$ or $e_i^{(2)}$ is an encrypted HTTP response with body g from fwd, then $\pi_1(g)$ is `script_fwd`.
5. If $e_i^{(1)}$ or $e_i^{(2)}$ is an HTTP(S) response with body g from a relying party, then it does not contain any `Location`, `Strict-Transport-Security` or `Set-Cookie` header and if $\pi_1(g)$ is a string representing a script, then $\pi_1(g)$ is either `script_rp` or `script_rp_redir`.
6. Neither $e_i^{(1)}$ nor $e_i^{(2)}$ are DNS responses from dns for domains unknown to the DNS server.
7. If $e_i^{(1)}$ or $e_i^{(2)}$ is an unencrypted HTTP response, then the message was sent by some attacker.

Definition 81 (Equivalence of Configurations). We call (S_1, E_1, N_1) and (S_2, E_2, N_2) α -equivalent iff there exists a set of proto-tags θ and a set of nonces H such that S_1 and S_2 are γ -equivalent under (θ, H) , E_1 and E_2 are β -equivalent under (θ, L, H) for $L := \bigcup_{a \in \theta} \text{loginSessionTokens}(a, S_1, S_2)$, and $N_1 = N_2$.

H.3 Privacy Proof

Theorem 3. *Every SPRESSO web system for privacy analysis is IdP-private.*

Let $\mathcal{SWS}^{priv} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0, \text{attacker})$ be an SPRESSO web system for privacy analysis.

To prove Theorem 3, we have to show that the SPRESSO web systems \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} are indistinguishable (according to Definition 73), where \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} are defined as follows: Let $\mathcal{SWS}_1^{priv} = (\mathcal{W}_1, \mathcal{S}, \text{script}, E^0, \text{attacker})$ and $\mathcal{SWS}_2^{priv} = (\mathcal{W}_2, \mathcal{S}, \text{script}, E^0, \text{attacker})$ with $b_1 := b(dr_1) \in \mathcal{W}_1$ and $b_2 := b(dr_2) \in \mathcal{W}_2$ challenge browsers. Further, we require $\mathcal{W} \setminus \{b\} = \mathcal{W}_1 \setminus \{b_1\} = \mathcal{W}_2 \setminus \{b_2\}$. We denote the following processes in \mathcal{SWS}^{priv} as in Definition 72:

- dns denotes the honest DNS server,
- fwd denotes the honest forwarder with domain `fwddomain`,
- r_1 denotes the honest relying party with domain dr_1 , and
- r_2 denotes the honest relying party with domain dr_2 .

Following Definition 69, to show the indistinguishability of \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} we show that they are indistinguishable under all schedules σ . For this, we first note that for all σ , there is only one run induced by each σ (as our web system, when scheduled, is deterministic). We now proceed to show that for all schedules $\sigma = (\zeta_1, \zeta_2, \dots)$, iff σ induces a run $\sigma(\mathcal{SWS}_1^{priv})$ there exists a run $\sigma(\mathcal{SWS}_2^{priv})$ such that $\sigma(\mathcal{SWS}_1^{priv}) \approx \sigma(\mathcal{SWS}_2^{priv})$.

We now show that if two configurations are α -equivalent, then the view of the attacker is statically equivalent.

Lemma 12. *Let (S_1, E_1, N_1) and (S_2, E_2, N_2) be two α -equivalent configurations. Then $S_1(\text{attacker}) \approx S_2(\text{attacker})$.*

Proof. From the α -equivalence of (S_1, E_1, N_1) and (S_2, E_2, N_2) it follows that $S_1(\text{attacker}) \equiv_{\theta} S_2(\text{attacker})$. From Condition 9 for γ -equivalence it follows that $\{k \mid \exists n : \text{enc}_s(\langle y, n \rangle, k) \in \theta\} \cap d_{\emptyset}(\bigcup_{i \in \{1,2\}, A \in \text{Web} \cup \text{Net}} S_i(A))$ (i.e., the attacker does not know any keys for the tags contained in its view), and therefore it is easy to see that the views are statically equivalent. \square

We now show that $\sigma(\mathcal{SWS}_1^{priv}) \approx \sigma(\mathcal{SWS}_2^{priv})$ by induction over the length of σ . We first, in Lemma 13, show that α -equivalence (and therefore, indistinguishability of the views of attacker) holds for the initial configurations of \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} . We then, in Lemma 14, show that for each configuration induced by a processing step in ζ , α -equivalence still holds true.

Lemma 13. *The initial configurations (S_1^0, E^0, N^0) of \mathcal{SWS}_1^{priv} and (S_2^0, E^0, N^0) of \mathcal{SWS}_2^{priv} are α -equivalent.*

Proof. We now have to show that there exists a set of proto-tags θ and a set of nonces H such that S_1^0 and S_2^0 are γ -equivalent under (θ, H) , $E_1^0 = E^0$ and $E_2^0 = E^0$ are β -equivalent under (θ, L, H) with $L := \bigcup_{a \in \theta} \text{loginSessionTokens}(a, S_1, S_2)$, and $N_1^0 = N_2^0 = N^0$.

Let $\theta = H = L = \emptyset$. Obviously, both latter conditions are true. For all parties $p \in \mathcal{W}_1 \setminus \{b_1\}$, it is clear that $S_1^0(p) = S_2^0(p)$. Also the states $S_1^0(b_1)$ and $S_2^0(b_2)$ are equal. Therefore, all conditions of Definition 79 are fulfilled. Hence, the initial configurations are α -equivalent. \square

Lemma 14. *Let (S_1, E_1, N_1) and (S_2, E_2, N_2) be two α -equivalent configurations of \mathcal{SWS}_1^{priv} and \mathcal{SWS}_2^{priv} , respectively. Let $\zeta = \langle ci, cp, \tau_{process}, cmd_{switch}, cmd_{window}, \tau_{script}, url \rangle$ be a web system command. Then, ζ induces a processing step in either both configurations or in none. In the latter case, let (S'_1, E'_1, N'_1) and (S'_2, E'_2, N'_2) be configurations induced by ζ such that*

$$(S_1, E_1, N_1) \xrightarrow{\zeta} (S'_1, E'_1, N'_1) \quad \text{and} \quad (S_2, E_2, N_2) \xrightarrow{\zeta} (S'_2, E'_2, N'_2).$$

Then, (S'_1, E'_1, N'_1) and (S'_2, E'_2, N'_2) are α -equivalent.

Proof. Let θ be a set of proto-tags and H be a set of nonces for which α -equivalence of (S_1, E_1, N_1) and (S_2, E_2, N_2) holds and let $L := \bigcup_{a \in \theta} \text{loginSessionTokens}(a, S_1, S_2)$, $K := \{k \mid \exists n : \text{enc}_s(\langle y, n \rangle, k) \in \theta\}$.

To induce a processing step, the ci -th message from E_1 or E_2 , respectively, is selected. Following Definition 80, we denote these messages by $e_i^{(1)}$ or $e_i^{(2)}$, respectively. We now differentiate between the receivers of the messages.

We first note that due to the α -equivalence, ζ either induces a processing step in both configurations or in none. We have to analyze the conditions stated in Corollary 1: The number of waiting events is the same in both configurations, and therefore, $(ci > |E_1|) \iff (ci > |E_2|)$. Further, the set of processes is the same (except for the browsers, which are exchanged but have the same IP addresses). Also, we have no processes that share IP addresses within each system. Therefore, if $cp \neq 1$ then it refers to no process in both runs (and no processing step can be induced). As we show below, if $e_i^{(1)}$ is delivered to b_1 then and only then $e_i^{(2)}$ is delivered to b_2 . Additionally, the window structure in both browsers is the same, and therefore, cmd_{window} either refers to a window that exists in both configurations or in none. There are no other cases that induce no processing step in either system.

We denote the induced processing steps by

$$(S_1, E_1, N_1) \xrightarrow[p_1 \rightarrow E_{\text{out}}^{(1)}]{\langle a_1, f_1, m_1 \rangle \rightarrow p_1} (S'_1, E'_1, N'_1) \text{ and}$$

$$(S_2, E_2, N_2) \xrightarrow[p_2 \rightarrow E_{\text{out}}^{(2)}]{\langle a_2, f_2, m_2 \rangle \rightarrow p_2} (S'_2, E'_2, N'_2).$$

Case $p_1 = \text{fwd}$: We know that one of the cases of Case 1 of Definition 80 must apply for $e_i^{(1)}$ and $e_i^{(2)}$. Out of these cases only Case 1a applies. Hence, $p_2 = \text{fwd}$.

In the forwarder relation (Algorithm 11), either Lines 9f. are executed in both processing steps or in none. It is easy to see that $E_{\text{out}}^{(1)} \equiv_{\theta} E_{\text{out}}^{(2)}$ (containing at most one event). For this new event all cases of Definition 80 except for Cases 2 and 1 hold trivially true.

(*): As both events are static except for IP addresses, the HTTP nonce, and the HTTPS key, there is no k contained in the input messages or in the state of fwd (except potentially in tags, from where it cannot be extracted), and the output messages are sent to f_1 or f_2 , respectively, they cannot contain any $l \in L$ or $k \in K$. Hence, Case 2 of Definition 80 holds true.

Both output events are constructed exactly the same out of their respective input events and Case 1a applies for the output events.

Therefore, E'_1 and E'_2 are β -equivalent under (θ, H, L) . As there are no changes to any state, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . No new nonces are chosen, hence, $N_1 = N'_1 = N_2 = N'_2$.

Case $p_1 = \text{dns}$: In this case, only Cases 1a, 1b and 1c of Definition 80 can apply. Hence, $p_2 = \text{dns}$. We note that (*) applies analogously in all cases.

In the first case, it is easy to see that $E_{\text{out}}^{(1)} \equiv_{\theta} E_{\text{out}}^{(2)}$. In the second case, it is easy to see that the DNS server only outputs empty events in both processing steps. In the third case, $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ are such that Case 1d of Definition 80 applies.

Therefore, E'_1 and E'_2 are β -equivalent under (θ, H, L) in all three cases. As there are no changes to any state in all cases, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . No new nonces are chosen, hence, $N_1 = N'_1 = N_2 = N'_2$.

Case $p_1 = r_1$: First, we consider cases that can never happen or are ignored in both processing steps. After this, we distinct several cases of HTTPS requests.

If $e_i^{(1)}$ is a DNS response, we know that $e_i^{(1)} \equiv_{\theta} e_i^{(2)}$, which implies $p_2 = r_1$. Only DNS responses from dns are processed by a relying party, other DNS responses are dropped without any state change. As the state of a relying party fulfills Condition 6 of Definition 79 (RPs only query domains unknown to dns) and both $e_i^{(1)}$ and $e_i^{(2)}$ fulfill Condition 6 of Definition 80 (there are no DNS responses from dns about domains unknown to dns), we have a contradiction. Hence, $e_i^{(1)}$ cannot be a DNS response.

If $e_i^{(1)}$ is an HTTP response, we know that $e_i^{(1)} \equiv_{\theta} e_i^{(2)}$, which implies $p_2 = r_1$. From Condition 7 of Definition 79, we know that relying parties always drop HTTP responses (without any state change).

If $e_i^{(1)}$ is any other message that is not a (properly) encrypted HTTP request, we have that $e_i^{(1)} \equiv_{\theta} e_i^{(2)}$, which implies $p_2 = r_1$. The relying party drops such messages in both processing steps (without any state change).

For the following, we note that a relying party never sends unencrypted HTTP responses.

There are four possible types of HTTP requests that are accepted by r_1 in Algorithm 9:

- $path = /$ (index page), Line 28,

- $path = /startLogin$ (start a login), Line 31,
- $path = /redir$ (redirect to IdP), Line 43, and
- $path = /login$ (login), Line 53.

From the cases in Definition 80, only two can possibly apply here: Case 1a and Case 1e. For both cases, we will now analyze each of the HTTP requests listed above separately.

Definition 80, Case 1a: $e_i^{(1)} \Rightarrow_{\theta} e_i^{(2)}$. This case implies $p_2 = r_1 = p_1$. As we see below, for the output events $E_{out}^{(1)}$ and $E_{out}^{(2)}$ (if any) only Case 1a of Definition 80 applies. This implies that the output events may not contain any HTTP nonce contained in H . As we know that the HTTP nonce of the incoming HTTP requests is not contained in H and the output HTTP responses (if any) of the RP reuses the same HTTP nonce, the nonce of the HTTP responses cannot be in H .

- $path = /$. In this case, the same output event is produced, i.e. $E_{out}^{(1)} = E_{out}^{(2)}$, and Condition 5 of Definition 80 holds true. Also, (*) applies. The remaining conditions are trivially fulfilled and E'_1 and E'_2 are β -equivalent under (θ, H, L) . As there are no changes to any state, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . No new nonces are chosen, hence, $N_1 = N'_1 = N_2 = N'_2$.

- $path = /startLogin$. The domains of the email addresses in both message bodies are either equivalent and registered to the DNS server (and hence, $wkCache$ contains a public key for this domain), or they are not contained in $wkCache$ (in both, $S_1(r_1)$ and $S_2(r_1)$). If they are unknown (i.e., not contained in $wkCache$), they are not registered in the DNS server. Nonetheless, in this case a DNS request is sent to dns . Then, the terms $E_{out}^{(1)}$ and $E_{out}^{(2)}$ contain a request matching Case 1a of Definition 80. As $E_{out}^{(1)}$ and $E_{out}^{(2)}$ are constructed such that besides IP addresses, a string, and a nonce, they only contain a term derived from the input events. In particular, they contain no $k \in K$ or $l \in L$ (**): As Condition 2 of Definition 80 applies for the input events, this condition also applies for the output events. Thus, E'_1 and E'_2 are β -equivalent under (θ, H, L) . The states $S'_1(r_1)$ is equal to $S_1(r_1)$ up to the subterm $pendingDNS$, and $S'_2(r_1)$ is equal to $S_2(r_1)$ up to the subterm $pendingDNS$. The subterm $pendingDNS$ only contains a new entry for a domain unknown to the DNS server. Hence, Condition 6 of Definition 79 holds. Thus, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . Exactly one nonce is chosen in both processing steps, and therefore $N'_1 = N'_2$.

If the domains of the email addresses are valid and registered in $wkCache$, then $SENDSTARTLOGINRESPONSE$ is called. In both processing steps, a tag is constructed exactly the same. The same HTTP response (which does not contain a $k \in K$ or a $l \in L$) is put in both $E_{out}^{(1)}$ and $E_{out}^{(2)}$. The first element of the response's body is not a string and therefore Condition 5 holds true. The tag is only created on r_1 in both runs and hence, θ does not have to be altered. Analogously to (**) we have that E'_1 and E'_2 are β -equivalent under (θ, H, L) . The subterm $loginSessions$ of the state of r_1 is extended exactly the same. Thus, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . In both processing steps exactly four nonces are chosen, and we have that $N'_1 = N'_2$.

- $path = /redir$. First, we note that there is no $l \in L$ contained in either m_1 or m_2 (by the Definition of β -equivalence). We further note that a relying party that receives an (encrypted) HTTP request for the path $/redir$ either (I) stops in Line 46 of Algorithm 9 with an empty output or (II) emits an HTTP response in Line 52. The state of the relying party is not changed in either case.

We now show that in both processing steps always the same cases apply. For $i \in \{1, 2\}$ and $body_i$ the body of the HTTPS request m_i , Case (II) applies for r_1 iff

$$S_i(r_1).loginSessions[body_i[loginSessionToken]] \neq \langle \rangle$$

(see Lines 44ff. of Algorithm 9).

From Condition 5 of Definition 79, we know that $S_2(r_1).loginSessions$ can be constructed from $S_1(r_1).loginSessions$ without removing the entry with the dictionary key $body_1[loginSessionToken]$ (as this key is not in L). Thus, both dictionaries either contain the same entry for the dictionary key $body_1[loginSessionToken]$ or they both contain no such entry. Hence, Case (II) applies in both processing steps or in none.

In both cases, exactly the same outputs are emitted (without containing any $l \in L$ or $k \in K$) and no state is changed and no new nonces are chosen. In both cases, the first element of the response body (if any) is $script_rp_redir$. We therefore trivially have α -equivalence of the new configurations.

- $path = /login$. This case can be handled analogously to the previous case with two exceptions:
 (A) First, there are two additional checks, the first in Line 54 of Algorithm 9 and the second in Line 64. We have to show that both checks each either simultaneously succeed or fail in both cases.

For the first check, it is easy to see that this follows from $m_1 \equiv_{\theta} m_2$.

As we have that $m_1 \equiv_{\theta} m_2$, and in particular $eia_1 := body_1[eia] \equiv_{\theta} body_2[eia] =: eia_2$, both, eia_1 and eia_2 have the same structure. If this structure does not match the expected structure (see Line 62f.), the checks in both processing steps fail.

If r_1 accepts the identity assertion, then we have that the tag, the email address and the forwarder domain must be equal in m_1 and m_2 as

$$\begin{aligned} & S_1(r_1).loginSessions[body_1[loginSessionToken]] \\ &= S_2(r_1).loginSessions[body_2[loginSessionToken]]. \end{aligned}$$

Hence, r_1 either accepts in both processing steps or in none.

- (B) If r_1 accepts, i.e., it does not stop with an empty message, then r_2 accepts. A nonce is chosen exactly the same in both processing steps. Hence, we have that $N'_1 = N'_2$.

Definition 80, Case 1e: $e_i^{(1)}$ is an HTTP(S) request from b_1 to r_1 and $e_i^{(2)}$ is an HTTP(S) request from b_2 to r_2 . This case implies $p_2 = r_2$.

We note that Condition 5 of Definition 80 holds for the same reasons as in the previous case. As the response is always addressed to the IP address of b_1 or b_2 , respectively, Condition 5 of Definition 80 is fulfilled.

As we see below, for the output events $E_{out}^{(1)}$ and $E_{out}^{(2)}$ (if any) only Case 1f of Definition 80 applies. This implies that the output events must contain an HTTP nonce contained in H . As we know that the HTTP nonce of the incoming HTTP requests is contained in H and the output HTTP responses (if any) of the RP reuses the same HTTP nonce, the nonce of the HTTP responses is in H .

- $path = /$. In this case, the output events produced (containing no $l \in L$ or $k \in K$ result in E'_1 and E'_2 being β -equivalent under (θ, H, L) according to Definition 80, Case 1f. As there are no changes to any state, we have that S'_1 and S'_2 are γ -equivalent under (θ, H) . No new nonces are chosen, hence, $N_1 = N'_1 = N_2 = N'_2$.
- $path = /startLogin$. As above, both email addresses in the input events either equivalent and their domain is known to the relying parties, or both email address domains are unknown. The latter case is analogue to above. Otherwise, `wkCache`, then `SENDSTARTLOGINRESPONSE` is called. In both processing steps, a tag is constructed the same up to the RP domain dr_1 or dr_2 , respectively. In both processing steps, an HTTP response is created. We denote the HTTP response generated by r_1 as m'_1 and the one generated by r_2 as m'_2 . We then have that

$$\begin{aligned} m'_1 &= enc_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, g_1 \rangle, k) \\ m'_2 &= enc_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, g_2 \rangle, k) \end{aligned}$$

with

$$\begin{aligned} g_1 &= \langle \langle \text{tagKey}, \nu_2 \rangle, \langle \text{loginSessionToken}, \nu_4 \rangle, \langle \text{FWDDomain}, S_1(r_1).\text{FWDDomain} \rangle \rangle \\ g_2 &= \langle \langle \text{tagKey}, \nu_2 \rangle, \langle \text{loginSessionToken}, \nu_4 \rangle, \langle \text{FWDDomain}, S_2(r_2).\text{FWDDomain} \rangle \rangle \end{aligned}$$

Obviously, m'_1 equals m'_2 . For $N_1 = N_2 = (n_1, n_2, \dots)$, we set $\theta' = \theta \cup \{enc_s(\langle y, n_1 \rangle, n_2)\}$, $N'_1 = N'_2 = (n_5, \dots)$ (as exactly four nonces are chosen in both processing steps), and $L' = L \cup \{n_4\}$. The receiver of both messages is the browser b_1 or b_2 , respectively. Obviously, it holds that $L' = \bigcup_{a \in \theta'} loginSessionTokens(a, S'_1, S'_2)$ and there exists an $l' \in L'$ such that $g_1[loginSessionToken] \equiv l'$. As Conditions 1f and 3 of Definition 80 hold, E'_1 and E'_2 are β -equivalent under (θ', H, L') . The subterm `loginSessions` of $S_1(r_1)$ is extended exactly the same as the subterm `loginSessions` of $S_2(r_2)$. Thus, we have that S'_1 and S'_2 are γ -equivalent under (θ', H) . (As mentioned above, in both processing steps exactly four nonces are chosen, and we have that $N'_1 = N'_2$.)

- $path = /redir$. By the definition of β -equivalence, the login session token l is the same. By the definition of γ -equivalence, we have that Algorithm 9 either (I) stops in both processing steps in Line 46 with an empty output or (II) (if $l \in L$) emits an HTTP response in both processing steps in Line 52.

From Condition 5 of Definition 79 we know that for $ls_1 := S_1(r_1).loginSessions[l]$ and $ls_2 := S_2(r_2).loginSessions[l]$ we have that $ls_1 \rightleftharpoons_{\theta} ls_2$.

We denote the HTTP response generated by r_1 as m'_1 and the one generated by r_2 as m'_2 . We then have that

$$\begin{aligned} m'_1 &= enc_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, g_1 \rangle, k) \\ m'_2 &= enc_s(\langle \text{HTTPResp}, n, 200, \langle \rangle, g_2 \rangle, k) \end{aligned}$$

with

$$\begin{aligned} g_1 &= \langle \text{script_rp_redir}, \langle \text{URL}, S, domain_1, /.well-known/spresso-login, params_1 \rangle \rangle \\ g_2 &= \langle \text{script_rp_redir}, \langle \text{URL}, S, domain_2, /.well-known/spresso-login, params_2 \rangle \rangle \end{aligned}$$

and $domain_1, domain_2, params_1, params_2$ derived exactly the same from ls_1 and ls_2 , respectively (and neither $params_1$ nor $params_2$ contains a key `loginSessionToken`). No keys $k \in K$ are contained in the output. Thus, the output events fulfill Condition 1f of Definition 80.

No state is changed and no new nonces are chosen. We therefore have α -equivalence of the new configurations.

- $path = /login$.

This case can be handled analogously to the previous case with two exceptions:

(A) First, there are two additional checks, the first in Line 54 of Algorithm 9 checks the origin header and the second in Line 64 checks the identity assertion.

As we know that $m_1 \simeq_{\theta} m_2$, we have that if the first check fails in r_1 then and only then it fails in r_2 . The same holds true for the second check.

If r_1 accepts the identity assertion, then we have that the email address must be equal in m_1 and m_2 as

$$\begin{aligned} &S_1(r_1).loginSessions[body_1[loginSessionToken]] \\ &= S_2(r_1).loginSessions[body_2[loginSessionToken]]. \end{aligned}$$

and we have that the identity assertion in g_1 is valid for r_1 , i.e., signed correctly and contains a tag for dr_1 , and thus, the identity assertion in g_2 is valid for r_2 . Hence, r_1 and r_2 either accept in both processing steps or in none.

(B) If r_1 accepts, i.e., it does not stop with an empty message, we know that r_2 accepts. A nonce is chosen exactly the same in both processing steps. Hence, we have that $N'_1 = N'_2$.

Case $p_1 = r_2$: This case is analogue to the case $p_1 = r_1$ above. Note that the Case 1e of Definition 80 cannot occur by definition.

Case $p_1 = b_1$: $\implies p_2 = b_2$

We now do a case distinction over the types of messages a browser can receive.

DNS response For the input events either Condition 1a of Definition 80 or Condition 1d apply. Therefore, the DNS request/response nonces in both events are equivalent up to RP domains under a set of proto-tags θ . From Condition 12b of Definition 79, we know that for a given nonce, there is either an entry in the dictionary *pendingDNS* in both browsers or in none. There are no entries under keys that are not nonces. Hence, both browsers either continue processing the incoming DNS response or stop with no state change and no output events in Line 55 of Algorithm 7. Further, we note that the resolved address contained in the DNS response has to be an IP address. From Condition 12b of Definition 79, we know that the protocol in both stored HTTP requests is the same. Therefore, the browsers either both choose a nonce (for HTTPS request) or none. There can now be two cases: (I) The IP addresses in both DNS responses are the same or (II) the IP address in m_1 is an IP address of r_1 and the IP address in m_2 is an IP address of r_2 . In both cases, the pending requests of the respective browsers are amended in such a way that they fulfill Condition 12c (as they fulfilled Condition 12b, which is essentially the same for HTTP(s) requests).

For $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$, we have that in Case (I) $E_{\text{out}}^{(1)} \rightleftharpoons_{\theta} E_{\text{out}}^{(2)}$ and hence Condition 1a of Definition 80 is fulfilled. In Case (II), the output messages fulfill Condition 1e.

From Condition 2 of Definition 80, we know that no $l \in L$ is contained in the DNS responses. Further, we know from Condition 1d of Definition 80 that if the IP addresses in the DNS responses differ, then they are responses for dr_1 and dr_2 , respectively. From Condition 12b of Definition 79, we know that only requests (prepared) for dr_1 and dr_2 , respectively, may contain a subterm $l \in L$. Hence, Condition 2 of Definition 79 holds true.

We also have that no $k \in K$ is contained in the response (with Condition 3 of Definition 79). There is also no $k \in K$ contained in the browser's pending HTTP requests, and therefore, there is none in the output events.

We have that S'_1 and S'_2 are γ -equivalent under (θ, H) , E'_1 and E'_2 are β -equivalent under (θ, H, L) , $N'_1 = N'_2$, and thus, the new configurations are α -equivalent.

HTTP response In this case, it is clear that the HTTP(s) response nonce, which has to match the nonce in the browser's `pendingRequests`, is either the same in both messages m_1 and m_2 or it contains a tag. If it contains a tag (with Condition 12c of Definition 79) or if it contains a nonce that is not in `pendingRequests` (which contains the same nonces for both browsers), both browsers stop and do not output anything or change their state.

We can now distinguish between two cases: In both browsers, (I) the *reference* that is stored along with the HTTP nonce is a window reference (in this case, the request was a "normal" HTTP(S) request), or (II) this reference is a pairing of a document nonce and an XHR reference chosen by the script that sent the request, which is an XHR. From Condition 12c of Definition 79 it is easy to see that no other cases are possible (in particular, the *reference* in both browsers is the same).

(I) In Case (I), we can distinguish between the following two cases:

- (a) The HTTP nonce in m_1 is in H : In this case, only Case 1f of Definition 80 can apply. We therefore have that there is no Location, Set-Cookie or Strict-Transport-Security header in the response, and that the responses m_1 and m_2 are equal up to proto-tags in θ . From Case 12c of Definition 79 we have that in both browsers b_1 and b_2 the encryption keys stored in `pendingRequests` are the same, that the expected sender in $e_i^{(1)}$ is r_1 and in $e_i^{(2)}$ is r_2 .

With this, we observe that both browsers either accept and successfully decrypt the messages and call the function `PROCESSRESPONSE`, or both browsers stop with not state change and no output event (in which case the α -equivalence is given trivially). In particular we note that the expected sender in both cases matches precisely the sender the message has (compare Case 1f of Definition 80).

In `PROCESSRESPONSE`, we see that no changes in the browsers' cookies are performed (as no cookies are in the response), the `sts` subterm is not changed, and no redirection is performed (as no Location header is present).

Now, new documents are created in each browser. These have the form

$$\langle \nu_1, location, referrer, script, scriptstate, \langle \rangle, \langle \rangle, \top \rangle$$

with

$$location = \langle URL, protocol, host, path, parameters \rangle .$$

Here, *script*, *scriptstate* are the same and *protocol*, *path*, *parameters* are taken from the requests, which means that these subterms are equal or term-equivalent up to proto-tags θ according to Case 12c of Definition 79. The host and the referrer are the same in both states up to exchange of domains, which can be dr_1 in b_1 and dr_2 in b_2 .

We note that if $k \in K$, then the request will not be of the correct form to be parsed into a document in the browser, and both browsers stop with an empty output and no state change.

The browser now attaches these newly created documents to its window tree, and we have to check that the Condition 12e of Definition 79 is satisfied.

As we have that both incoming messages were encrypted messages (see Case 7 of Definition 80) and both messages come from r_1 and r_2 , respectively, and therefore *script* is either `script_rp` or `script_rp_redir` (see Case 5 of Definition 80) we have to check Conditions 12(e)iv and 12(e)v of Definition 79 in particular.

The *scriptstate* is initially equal and may contain a subterm $l \in L$ (as we know from HTTP nonce in m_1 being in H that the host of this document is dr_1 in b_1 and dr_2 in b_2), and the script inputs are empty. The

document's referer is constructed from the referer header of the request, which is equal in both cases or has the host dr_1 in b_1 and dr_2 in b_2 .

To sum up, γ -equivalence under (θ, H) is preserved. α -equivalence is preserved as no output event is generated and the exact same number of nonces are chosen.

- (b) The HTTP nonce in m_1 is not in H : In this case we have that $e_i^{(1)} \Rightarrow_{\theta} e_i^{(2)}$ (Case 1a of Definition 80), and that the HTTP nonces, senders, encryption keys (if any) and original requests in the pending requests of both browsers are either equal or equivalent up to proto-tags θ . There can be no $k \in K$ as a subterm (except in tags) of the input.

With this, we observe that both browsers either accept and successfully decrypt the messages and call the function PROCESSRESPONSE, or both browsers stop with no state change and no output event (in which case the α -equivalence is given trivially). In particular we note that the expected sender in both cases matches precisely the sender of the message (as it is equal).

If there is a Set-Cookie header in one of the responses, a new entry in the cookies of each browsers is created (which obviously is term-equivalent up to θ , and therefore is in compliance with the requirements for γ -equivalence). The same holds true for any Strict-Transport-Security headers.

Now, if there is a Location header in m_1 (and therefore also in m_2), a new request is generated and stored under the pending DNS requests, and a DNS request is sent out. The new HTTP(S) requests contains the method, body, and Origin header of the original request (which were equivalent up to proto-tags θ), where the Origin header is amended by the host and protocol of the original request.

Also, we know from $e_i^{(1)} \Rightarrow_{\theta} e_i^{(2)}$ that neither event may contain a subterm $l \in L$ or $k \in K$. Hence, the transferred (initial) scriptstate (or a request generated by a Location header, see below) cannot contain a subterm $l \in L$ or $k \in K$.

Now, assuming that the domain in the Location header was not CHALLENGE, then the new request is term-equivalent under θ between both browsers. A new DNS request is generated (which conforms to Condition 1a of Definition 79). It is sent out and the HTTP request is stored in the pending DNS requests of each browser. It is clear that in this case, the conditions for γ -equivalence under (θ, H) (in particular, Condition 12b) and β -equivalence under (θ, H, L) are satisfied. The same number of nonces is chosen. Altogether, α -equivalence is given.

If, however, the domain is CHALLENGE (and the browser has not started a request to CHALLENGE before; in this case the browser would behave as above), then the domain is dr_1 in b_1 and dr_2 in b_2 . In particular, in the resulting requests, the Host header is exchanged in this way. For alpha equivalence to hold for the new configuration, we have $H' = H \cup \{n\}$, where n is the nonce chosen for the HTTP(S) request. A new DNS request is generated (which in this case conforms to Condition 1b of Definition 79). Therefore, we have γ -equivalence under (θ, H') and β -equivalence under (θ, H', L) . The same number of nonces is chosen, and we indeed have α -equivalence.

If there is no Location header in m_1 (and therefore none in m_2), a new document is constructed just as in the case when the nonce in m_1 is in H .

The scriptstate is initially equal, and the script inputs are empty. The document's referer is constructed from the referer header of the request, which is equal in both cases (up to proto-tags in θ).

To sum up, γ -equivalence under (θ, H) is preserved in this case as well. α -equivalence is preserved as no output event is generated and the exact same number of nonces are chosen.

- (II) In Case (II), i.e., the response is a response to an XHR, we have that *reference* is a tuple, say, *reference* = $\langle docnonce, xhrref \rangle$, and we again distinguish between the two cases as above:

- (a) The HTTP nonce in m_1 is in H : In this case, only Case 1f of Definition 80 can apply. We therefore have that there is no Location, Set-Cookie or Strict-Transport-Security header in the response, and that the responses m_1 and m_2 are equal up to proto-tags in θ . From Case 12c of Definition 79 we have that in both browsers b_1 and b_2 the encryption keys stored in pendingRequests are the same and that the expected sender in $e_i^{(1)}$ is r_1 and in $e_i^{(2)}$ is r_2 .

With this, we observe that both browsers either accept and successfully decrypt the messages and call the function PROCESSRESPONSE, or both browsers stop with not state change and no output event (in

which case the α -equivalence is given trivially). In particular we note that the expected sender in both cases matches precisely the sender of the message (compare Case 1f of Definition 80).

In PROCESSRESPONSE, we see that no changes in the browsers' cookies are performed (as no cookies are in the response), the `sts` subterm is not changed, and no redirection is performed (as no Location header is present).

A new input is constructed for the document that is identified by *docnonce*. We note that such a document exists either in both browsers or in none (in which, again, both browsers stop with no output or state change). As the input events may contain a subterm $l \in L$ (as we know from HTTP nonce in m_1 being in H that the host of this document is dr_1 in b_1 and dr_2 in b_2), the constructed scriptinput may also contain a subterm $l \in L$. The same holds true for keys $k \in K$.

For $j \in \{1, 2\}$, we have that the `scriptinput` term for the document in b_j is $\langle \text{XMLHTTPREQUEST}, g_j.\text{body}, x\text{href} \rangle$, where g_j is the HTTP body of m_j . With $g_1 \Rightarrow_\theta g_2$ and $x\text{href} \in \mathcal{N} \cup \{\perp\}$, it is easy to see that the resulting `scriptinput` term of the document is term-equivalent under proto-tags θ (as it was before). This satisfies γ -equivalence on the new browser state.

No output event is generated, and no nonces are chosen. Therefore we have α -equivalence on the new configuration.

- (b) The HTTP nonce in m_1 is not in H : In this case we have that $e_i^{(1)} \Rightarrow_\theta e_i^{(2)}$ (Case 1a of Definition 80), and that the HTTP nonces, senders, encryption keys (if any) and original requests in the pending requests of both browsers are either equal or equivalent up to proto-tags θ .

With this, we observe that both browsers either accept and successfully decrypt the messages and call the function PROCESSRESPONSE, or both browsers stop with not state change and no output event (in which case the α -equivalence is given trivially). In particular we note that the expected sender in both cases matches precisely the sender the message has (as it is equal).

If there is a Set-Cookie header in one of the responses, a new entry in the cookies of each browsers is created (which obviously is term-equivalent up to θ , and therefore is in compliance with the requirements for γ -equivalence). The same holds true for any Strict-Transport-Security headers.

Now, if there is a Location header in m_1 (and therefore also in m_2), both browsers stop with not state change and no output event (in which case the α -equivalence is given trivially), as XHR cannot be redirected in the browser.

If there is no Location header in m_1 (and therefore none in m_2), a new input is constructed for the document that is identified by *docnonce*. We note that such a document exists either in both browsers or in none. For $j \in \{1, 2\}$, we have that the `scriptinput` for the document in b_j is $\langle \text{XMLHTTPREQUEST}, g_j.\text{body}, x\text{href} \rangle$, where g_j is the HTTP body of m_j . With $e_i^{(1)} \Rightarrow_\theta e_i^{(2)}$ (which may not contain a subterm $l \in L$ or $k \in K$), it is easy to see that the resulting `scriptinput` term of the document is term-equivalent under proto-tags θ (as it was before). This satisfies γ -equivalence on the new browser state.

No output event is generated, and no nonces are chosen. Therefore we have α -equivalence on the new configuration.

TRIGGER We now distinguish between the possible values for cmd_{switch} .

1 (trigger script): In this case, the script in the window indexed by cmd_{window} is triggered. Let j be a pointer to that window.

We first note that such a window exists in b_1 iff it exists in b_2 and that $S_1(b_1).j.\text{script} \equiv S_2(b_2).j.\text{script}$. We now distinguish between the following cases, which cover all possible states of the windows/documents:

1. $S_1(b_1).j.\text{origin} \in \{\langle dr_1, S \rangle, \langle dr_2, S \rangle\}$ and $S_1(b_1).j.\text{script} \equiv \text{script}_{\text{rp}}$.

Similar to the following scripts, the main distinction in this script is between the script's internal states (named `q`). With the term-equivalence under proto-tags θ we have that either $S_1(b_1).j.\text{scriptstate.q} = S_2(b_2).j.\text{scriptstate.q}$ or the script's state contains a tag and is therefore in an illegal state (in which case the script will stop without producing output or changing its state).

We can therefore now distinguish between the possible values of $S_1(b_1).j.\text{scriptstate.q} = S_2(b_2).j.\text{scriptstate.q}$:

start: In this case, the script chooses one nonce in both processing steps and creates an XHR addressed to its own origin, which is in both cases either (a) equal and $\langle dr_1, S \rangle$ or $\langle dr_2, S \rangle$ or it is (b) $\langle dr_1, S \rangle$ in b_1 and $\langle dr_2, S \rangle$ in b_2 . The path is the (static) string `/startLogin`. The script saves the freshly chosen nonce (referencing the XHR) and a (static) value for `q` in its `scriptstate`. We note that if a $k \in K$ is contained in the script's state, it is never sent out in this state.

In Case (a), we have that the command is term-equivalent under proto-tags θ and hence, the browser emits a DNS request which is term-equivalent, and appends the XHR in `pendingDNS`. Hence, we have γ -equivalence under (θ, H) for the new states, β -equivalence under (θ, H, L) for the new events, and α -equivalence for the new configuration.

In Case (b), we have that the prepared HTTP request is δ -equivalent under θ , and is added to `pendingDNS` in the browser's state, we set $H' := H \cup \{n\}$ with n being the nonce that the browser chooses for λ_1 . The browser emits a DNS request that fulfills Condition 1b of Definition 80. Therefore, we have γ -equivalence under (θ, H') for the new states. As the request added to `pendingDNS` in the browser's state fulfills Condition 12b, we have β -equivalence under (θ, H', L) for the new events, and α -equivalence for the new configuration.

expectStartLoginResponse: In this case, the script retrieves the result of an XHR from `scriptinputs` that matches the reference contained in `scriptstate`. From Condition 12(e)iv of Definition 79 we know that all results from XHRs in `scriptinput` are term-equivalent up to θ and that `scriptstate` is term-equivalent up to θ . Hence, in both browsers, both scripts stop with an empty command or both continue as they successfully retrieved such an XHR.

Now, a URL is constructed (exactly the same) for an (HTTPS) origin that is the origin of the document. We have to distinguish between two cases: Either the origin is (i) equal in b_1 and b_2 , or (ii) the origin is $\langle dr_1, S \rangle$ in b_1 and $\langle dr_2, S \rangle$ in b_2 . In the first case (i), no subterm $l \in L$ is contained in `scriptinput` and hence, no such subterm is contained in the constructed URL. In the second case (ii), however, we have that such a subterm may be contained in `scriptinput`. But as the URL commands the browser to prepare a request to dr_1 and dr_2 , respectively, the request may be stored in `pendingDNS` of the browser's state.

To store the prepared HTTP request in `pendingDNS`, the browser chooses a nonce n . We construct the set H' as follows: In (i) $H' := H$ and in (ii) $H' := H \cup \{n\}$. Thus, Condition 12b of Definition 79 holds true under (θ, H') .

In Case (i) we also have that no $k \in K$ is written into `scriptstate.tagKey`. Otherwise, a $k \in K$ may be written there. In no case, a $k \in K$ is contained in the output event or generated HTTP request (as `scriptstate.tagKey` is not used to create such event or request).

The output events of both browsers are either a DNS request that is equal in (i) or a DNS request that matches Condition 1b of Definition 80.

We now have that S'_1 and S'_2 are γ -equivalent under (θ, H') , E'_1 and E'_2 are β -equivalent under (θ, H', L) , and as exactly the same number of nonces is chosen, we have that the new configuration is α -equivalent.

expectFWDReady: In this case, the script retrieves the result of a `postMessage` from `scriptinputs`. As we know that $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$ and that for all matching `postMessages` that they also have to be term-equivalent up to θ and that the window structure is equal in both browsers, we have that either the same `postMessage` is retrieved from `scriptinputs` or none in both browsers.

The script now constructs a `postMessage` that is sent to exactly the same window in both browsers and that requires that the receiver origin has to be $\langle fwdomain, S \rangle$. The `postMessage` is only sent to this origin, we have that γ -equivalence cannot be violated even if a $k \in K$ is contained in the `postMessage` (as there are no constraints concerning K in the script inputs of this origin).

We now have that S'_1 and S'_2 are γ -equivalent under (θ, H) , E'_1 and E'_2 are β -equivalent under (θ, H, L) , and as exactly the same number of nonces is chosen, we have that the new configuration is α -equivalent.

expectEIA: In this case, the script retrieves the result of a `postMessage` from `scriptinputs`. As we know that $S_1(b_1).j.scriptstate \equiv_{\theta} S_2(b_2).j.scriptstate$ and that for all matching `postMessages` that

they also have to be term-equivalent up to θ and that the window structure is equal in both browsers, we have that either the same `postMessage` is retrieved from `scriptinputs` or none in both browsers.

The script chooses one nonce in both processing steps and creates an XHR addressed to its own origin, which is in both cases either (a) equal and $\langle dr_1, S \rangle$ or $\langle midr_2, S \rangle$ or it is (b) $\langle dr_1, S \rangle$ in b_1 and $\langle midr_2, S \rangle$ in b_2 . The path is the (static) string `/login`. The script saves the freshly chosen nonce (referencing the XHR) and a (static) value for `q` in its `scriptstate`.

In Case (a), we have that the command is term-equivalent under proto-tags θ and hence, the browser emits a DNS request which is term-equivalent, and appends the XHR in `pendingDNS`. Hence, we have γ -equivalence under (θ, H) for the new states, β -equivalence under (θ, H, L) for the new events, and α -equivalence for the new configuration.

For Case (b), we note that for $j \in \{1, 2\}$, the body g_j of the prepared HTTP request may contain an (encrypted) identity assertion such that

$$g_j[\text{eia}] \sim \text{enc}_s(\text{sig}(\langle \text{enc}_s(\langle dr_j, * \rangle, *) \rangle, *, \text{fwddomain}), *, *).$$

As the receiver of this message is always determined to be dr_j (in b_j) and the Origin header is set accordingly, we have that the prepared HTTP request is δ -equivalent under θ . The (prepared) request is added to `pendingDNS` in the browser's state, we set $H' := H \cup \{n\}$ with n being the nonce that the browser chooses for λ_1 . The browser emits a DNS request that fulfills Condition 1b of Definition 80. In no case is a $k \in K$, which can only be stored in `scriptstate.tagKey` used to construct either output events or state changes. Therefore, we have γ -equivalence under (θ, H') for the new states. As the request added to `pendingDNS` in the browser's state fulfills Condition 12b, we have β -equivalence under (θ, H', L) for the new events, and α -equivalence for the new configuration.

2. $S_1(b_1).j.\text{origin} \in \{\langle dr_1, S \rangle, \langle dr_2, S \rangle\}$ and $S_1(b_1).j.\text{script} \neq \text{script_rp}$. It immediately follows that $S_1(b_1).j.\text{script} \equiv \text{script_rp_redir}$ in this case. This script always outputs the same command to the browser: it commands to navigate its window to a URL saved as the script's `scriptstate`, which is term-equivalent under proto-tags between the two browsers. As the HTTP request that is generated (and then stored in `pendingDNS`) contains neither an Origin nor a Referer header, they are term-equivalent under θ if the domain of this HTTP request is not CHALLENGE²³ and δ -equivalent otherwise.

The resulting states of both browsers are therefore γ -equivalent under θ .

In both cases (challenged or not), we have that $E_{\text{out}}^{(1)} \rightleftharpoons_{\theta} E_{\text{out}}^{(2)}$ and hence Condition 1a of Definition 80 is fulfilled, or the output messages fulfill Condition 1e.

Clearly, no $k \in K$ is contained in the script's state, in the generated DNS request, or the HTTP request.

In both processing steps, exactly the same number of nonces is chosen. We therefore have α -equivalence.

3. $S_1(b_1).j.\text{origin} = \langle \text{fwddomain}, S \rangle$.

It immediately follows that $S_1(b_1).j.\text{script} \equiv \text{script_fwd}$ in this case.

As above, we have that either $S_1(b_1).j.\text{scriptstate.q} = S_2(b_2).j.\text{scriptstate.q}$ or the script's state contains a tag and is therefore in an illegal state (in which case the script will stop without producing output or changing its state).

With the equivalence of the window structures we have that the `target` variable in the algorithm of `script_fwd` in both runs points to a window containing the same script in both runs.

We can now distinguish between the possible values of $S_1(b_1).j.\text{scriptstate.q} = S_2(b_2).j.\text{scriptstate.q}$:

start In this case, a `postMessage` with exactly the same contents is sent to the same window. We therefore trivially have γ -equivalence under (θ, H) on the states in this case. No output events are generated, and no nonces are used. Therefore, α -equivalence holds on the new states.

expectTagKey In this case, for any change in the state to occur, a `postMessage` containing some term under the (dictionary) key `tagKey` sent from exactly the same window has to be in `scriptinputs`. From Condition 12(e)vi of Definition 79 we know that in `scriptinputs` either such a `postMessage` exists in both browsers or in none.

²³This also applies when the browser challenged before, i.e., `challenge` in the browser's state is \perp .

As the *tag* contained in the `postMessages` is term-equivalent under proto-tags θ , we have that the RP domain inside the tag is either the same in both processing steps or dr_1 in b_1 and dr_2 in b_2 . Additionally, *eia* (if contained in the URL parameters in the location of the script) is term-equivalent under proto-tags θ . It follows that the resulting `postMessage`, which contains *eia*, is either delivered to the receiver (which is either equal in both browsers or dr_1 in b_1 and dr_2 in b_2). Additionally, no $k \in K$ can be contained in the *eia*, as it is taken from the document's location. In any case, the resulting browser states are γ -equivalent under (θ, H) .

As no output events are produced, we have that E'_1 and E'_2 are β -equivalent under (θ, H, L) . As exactly the same number of nonces are chosen (in fact, no nonces are chosen), we have that the resulting configuration is α -equivalent.

4. $S_1(b_1).j.\text{origin} \notin \{\langle dr_1, S \rangle, \langle dr_2, S \rangle, \langle \text{fwdomain}, S \rangle\}$.

Here, we assume that the script in this case is the attacker script R^{att} , as it subsumes all other scripts. We first observe, that its “view”, i.e., the input terms it gets from the browser, is term-equivalent up to proto-tags θ between (the scripts running in) $S_1(b_1)$ and $S_2(b_2)$. From the equivalence definition of states (Definition 79) we can see that:

- The window tree has the same structure in both processing steps. All window terms are equal (up to their `documents` subterm). All same-origin documents contain only subterms that are term-equivalent up to θ (again, up to their `subwindows` subterms). All non-same-origin documents become limited documents and therefore are equal (up to the `subwindows`, limited documents only contain the `subwindows` and the document nonce).
- The subterms `cookies`, `localStorage`, `sessionStorage`, `scriptstate`, and `scriptinputs` are term-equivalent up to θ .
- The subterms `ids` and `secrets` are equal.
- There is not $k \in K$ as a subterm (except as keys for tags) in this view. We therefore have that no such term can be contained in the output command of the script, or in the new `scriptstate`.

As the input of the script as a whole is term-equivalent up to θ , does not contain any placeholders in V_{script} , and does not contain a key for any tag in θ , we have that the output of the script, i.e., `scriptstate'`, `cookies'`, `localStorage'`, `sessionStorage'`, `command'`, must be term-equivalent up to proto-tags θ (in particular, the same number of nonces is replaced in both output terms in both processing steps). Note that the first element of the command output must be equal between the two browsers (as it must be string) or otherwise the browsers will ignore the command in both processing steps.

Analogously, we see that the input does not contain any subterm $l \in L$.

We can now distinguish the possible commands the script can output (again, all parameters for these commands must be term-equivalent under θ):

- (a) Empty or invalid command: In this case, the browser outputs no message and its state is not changed. α -equivalence is therefore trivially given.
- (b) $\langle \text{HREF}, \text{url}, \text{hrefwindow}, \text{noreferrer} \rangle$: Here, the browser calls `GETNAVIGABLEWINDOW` to determine the window in which the document will be loaded. Due to the synchronous window structure between the two browsers, the result will be the same in both processing steps (which may include creating a new window with a new nonce).

Now, a new HTTP(S) request is assembled from the URL. A `Referer` header is added to the request from the document's current `location` (which is term-equivalent under θ) and given to the `SEND` function. There, if the `host` part of the URL is `CHALLENGE`, it will be replaced by dr_1 in b_1 and by dr_2 in b_2 . (In this case, the α -equivalence in the following holds for $H' := H \cup \{n\}$, where n is the nonce of the generated HTTP request. Otherwise, it holds for $H' := H$.) Afterwards, for domains that are in the `sts` subterm of the browser's state, the request will be rewritten to HTTPS. Any cookies for the domain in the requests are added. Note that both latter steps never apply to requests to dr_1 or dr_2 as per definition, there are no entries for these domains in `sts` and `cookies`. The requests, which are δ -equivalent under θ are added to the pending DNS requests and fulfill Condition 12b of Definition 79. A DNS request is created in accordance with Condition 1b or Condition 1a of Definition 80. The same number of nonce is chosen in both processing steps, and therefore α -equivalence holds.

- (c) $\langle \text{IFRAME}, \text{url}, \text>window \rangle$ This case is completely parallel to Case 4b.

- (d) $\langle \text{FORM}, url, method, data, hrefwindow \rangle$ This case is parallel to Case 4b, except that an Origin header is added. Its properties are the same as those of the Referer header in Case 4b.
- (e) $\langle \text{SETSCRIPT}, window, script \rangle$ In this case, the same document is manipulated in both processing steps in the same way. Note that only same-origin documents, i.e., attacker documents, can be manipulated. No output event is generated, and no nonces are chosen. α -equivalence is given trivially.
- (f) $\langle \text{SETSCRIPTSTATE}, window, scriptstate \rangle$ This case is parallel to Case 4e.
- (g) $\langle \text{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$ This case is parallel to Case 4b with the addition of the Origin header (see Case 4d) and the addition of a reference parameter, which is transferred into pendingDNS inside the browser (*xhrreference*). Therefore, for γ -equivalence, it is important to note that this reference can only be a nonce (and therefore is equal in both processing steps). Otherwise, the browser stops in both processing steps.
- (h) $\langle \text{BACK}, window \rangle$, $\langle \text{FORWARD}, window \rangle$, and $\langle \text{CLOSE}, window \rangle$ If the script outputs one of these commands, in both processing steps, the browsers will be manipulated in exactly the same way. No output events are generated, and no nonces are chosen.
- (i) $\langle \text{POSTMESSAGE}, window, message, origin \rangle$ In this case, a term containing *message* (term-equivalent under θ) is added to a document's *scriptinput* term. If the *origin* is \perp , the same term will be added to the same document in both processing steps. Otherwise, the term may only be added to one document (if, for example, the origin is $\langle dr_1, S \rangle$ and the target documents in both browsers have the domain dr_1 and dr_2 , respectively). In this case, however, the equivalence defined on the *scriptinputs* is preserved. This would only be possible for *script_rp* and only if the sender origin was $\langle fwddomain, S \rangle$.

2 (navigate to URL): In this case, a new window is opened in the browser and a document is loaded from *url*.

The states of both browsers are changed in the same way except if the domain of the URL is CHALLENGE. In both cases, a new (at this point empty) window is created and appended the *windows* subterm of the browsers. This subterm is therefore changed in exactly the same way.

A new HTTP request is created and appended to pendingDNS. The generated requests in both processing steps can only differ in the host part iff the domain is CHALLENGE. In this case, in b_1 the domain is replaced by dr_1 and in b_2 by dr_2 and the α -equivalence in the following holds for $H' := H\{n\}$, where n is the nonce of the generated HTTP request. In both cases, the Condition 12b of Definition 79 is satisfied.

The request cannot contain any $l \in L$ or $k \in K$.

The generated DNS requests are equivalent under Condition 1b or Condition 1a of Definition 80.

In both processing steps, three nonces are chosen.

Therefore, we have α -equivalence for (S'_1, E'_1, N'_1) and (S'_2, E'_2, N'_2) .

3 (reload document): Here, an existing document is retrieved from its original location again. From the definition of γ -equivalence under (θ, H) we can see that whatever document is reloaded, its location is either (I) term-equivalent under θ , or (II) it is term-equivalent under θ except for the domain, which is dr_1 in b_1 and dr_2 in b_2 .

We note that in either case, the requests are constructed from the location and referrer properties of the document that is to be reloaded, and therefore, cannot contain any $k \in K$.

In Case (I), we note that the domain cannot be CHALLENGE. If the document is reloaded, the same DNS request is issued in both browsers (therefore, β -equivalence under (θ, H, L) is given), and an entry is added to the pending DNS messages such that we have γ -equivalence under (θ, H) . The same number of nonces is chosen in both runs, and we have α -equivalence.

Case (II) is similar, but we have $H' := H \cup \{n\}$, where n is the nonce of the HTTP request that is added to the pending DNS entries. Then we have γ -equivalence under (θ, H') . Again, the same number of nonces is chosen and we have α -equivalence.

Other Any other message is discarded by the browsers without any change to state or output events.

Case p_1 is some attacker:

Here, only Case 1a from Definition 80 can apply to the input events, i.e., the input events are term-equivalent under proto-tags θ . This implies that the message was delivered to the same attacker process in both processing steps. Let A be that attacker process. With Case 10 of Definition 79 we have that $S_1(A) \equiv_{\theta} S_2(A)$ and with Case 9 and Case 3 of

Definition 80 it follows immediately that the attacker cannot decrypt any of the tags in θ in its knowledge. Further, in the attacker's state there are no variables (from V_{process}).

With the output term being a fixed term (with variables) $\tau_{\text{process}} \in \mathcal{T}_{\mathcal{N}}(\{x\} \cup V_{\text{process}})$ and x being $S_1(A)$ or $S_2(A)$, respectively, and there is no subterm $l \in L$ contained in either $S_1(A)$ or $S_2(A)$ (Condition 11 of Definition 79), it is easy to see that the output events are β -equivalent under θ , i.e., $E_{\text{out}}^{(1)} \rightleftharpoons_{\theta} E_{\text{out}}^{(2)}$, there are not $k \in K$ contained in the output events (except as encryption keys for tags) and the used nonces are the same, i.e., $N'_1 = N'_2$. The new state of the attacker in both processing steps consists of the input events, the output events, and the former state of the event, and, as such, is β -equivalent under proto-tags θ . Therefore we have α -equivalence on the new configurations. □

This proves Theorem 3. ■