Privacy-Preserving Web Single Sign-On: Formal Security Analysis and Design

Von der Fakultät 5 (Informatik, Elektrotechnik und Informationstechnik) der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Guido Tim Roman Schmitz

aus Trier

Hauptberichter:	Prof. Dr. Ralf Küsters
Mitberichter:	Dr. Karthikeyan Bhargavan
Tag der mündlichen Prüfung:	30.10.2019

Institut für Informationssicherheit (SEC) der Universität Stuttgart

2019

Acknowledgements

I would like to gratefully thank my advisor Ralf Küsters for giving me the opportunity to write this thesis. His excellent support made this thesis possible, and he enriched my research with many fruitful discussions and invaluable advice.

Also, I would like to thank all my colleagues who accompanied me in Trier and Stuttgart. They have created a great environment, not only for work but also for many other activities. In particular, I would like to thank Daniel Fett, with whom I have worked closely over the years and started many exciting and on-going projects.

Furthermore, I would like to thank the Deutsche Forschungsgemeinschaft for partially supporting my research through Grants KU 1434/10-1 and KU 1434/10-2.

Last but not least, I would like to thank Astrid and my family for their continuous support and encouragement.

Contents

List of Figures

Notation in protocol sequences. In figures, where we show detailed protocol sequences, we use the following notation:

→ HTTPS requests	⇢ XHRs requests (over HTTPS)	⇢ postMessages
← HTTPS responses	⇠ XHRs responses (over HTTPS)	⇢ browser commands (e.g., open a window)

List of Tables

List of Algorithms

List of Abbreviations

AD Authentication Dialog

API Application Programming Interface

CAP Certificate Assertion Pair

CIF Communication Iframe

CORS Cross-Origin Resource Sharing

DANE DNS-based Authentication of Named Entities

DNS Domain Name System

DNSSEC DNS Security

DY Dolev-Yao

FWD Forwarder

FWD-Doc A Document Under the Origin of an FWD

HTML Hypertext Markup Language

HTML5 HTML Version 5

HTTP Hypertext Transfer Protocol

HTTP/1.1 HTTP Version 1.1

HTTPS HTTP Over TLS

IA Identity Assertion

id Identity

IdP Identity Provider

IdP-Doc A Document Under the Origin of an IdP

IETF Internet Engineering Task Force

IP Internet Protocol

LD Login Dialog

LPO Server at the Domain `login.persona.org`

PIF Provisioning Iframe

PKI Public-Key Infrastructure

RFC Request for Comments

RP Relying Party

RP-Doc A Document Under the Origin of an RP

SAML Security Assertion Markup Language

SMS Short Message Service

SMT Satisfiability Modulo Theory

SRI Subresource Integrity

SSO Single Sign-On

STS Strict Transport Security

TCP Transmission Control Protocol

TLS Transport Layer Security

UC User Certificate

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WHATWG Web Hypertext Application Technology Working Group

WIM Web Infrastructure Model

WWW World Wide Web

XHR XMLHttpRequest

XRI Extensible Resource Identifier

XSRF Cross-Site Request Forgery

XSS Cross-Site Scripting

Abstract

Web-based single sign-on (SSO) systems enable Web sites, so-called relying parties (RPs), to outsource user authentication to other entities, so-called identity providers (IdPs). Such systems are widely deployed in the Web, e.g., Facebook Login or Google Sign-in. RPs do not need to maintain authentication data of their users, and users can log in at RPs in a convenient way.

Fundamental to SSO is security: The SSO protocol must not permit an attacker to impersonate anyone else, nor must it allow a false identity to be imposed on anyone. If this is not the case, attacks are possible that have devastating effects on the security of RPs and their users.

While aiming at security, most SSO systems, however, neglect privacy. IdPs can track their users as they (by design) learn at which RP a user logs in. This lack of privacy allows IdPs to create extensive user profiles and might cause some users not to use SSO at all. Moreover, IdPs are enabled to decide ad-hoc whether they allow a user to log in at a specific RP. Therefore, privacy-preserving systems, which do not reveal to IdPs to which RP a user would like to log in or has logged in, are highly desirable in many situations. The design of such systems, however, is very challenging because privacy can easily be compromised. So far, only one SSO system has been proposed with this kind of privacy in mind: Mozilla's BrowserID (a.k.a. Mozilla Persona).

In this thesis, we use the Web Infrastructure Model (WIM) to analyze the security of SSO protocols. The WIM is the most comprehensive formal model of the Web infrastructure to date, which applies to a wide range of Web applications and standards. We also extend the WIM to be able to analyze privacy. We use the extended WIM to, for the first time, carry out a systematic and rigorous formal analysis of privacy for Web SSO systems.

Using our approach, we analyze the Web SSO system BrowserID. As a result of this first rigorous analysis of an SSO system in the Web infrastructure, we find severe attacks. These attacks not only affect the security of BrowserID but also show that BrowserID's unique privacy claim does not hold. We propose fixes for BrowserID and prove that the fixed system provides security. Regarding privacy, we show that BrowserID, unfortunately, is broken beyond repair.

Inspired by BrowserID's goal, we propose the first privacy-preserving Web SSO system, called SPRESSO (for Secure Privacy-REspecting Single Sign-On). SPRESSO is easy to use, decentralized and based solely on native Web features. We design SPRESSO within the WIM right from the start and prove that SPRESSO satisfies strong security and privacy guarantees.

Kurzzusammenfassung

Webbasierte Single-Sign-On-Systeme (SSO) ermöglichen es Webdiensten, so genannten Relying Parties (RPs), Benutzerauthentifizierung an Dritte, so genannte Identity Provider (IdPs), auszulagern. Solche Systeme, z.B. Facebook Login oder Google Sign-In, sind weit verbreitet. RPs müssen weniger kritische Daten speichern, und Benutzer können sich bequem bei RPs anmelden.

Sicherheit ist für SSO essentiell: Es darf nicht möglich sein, sich als jemand anderes auszugeben, noch darf jemandem eine falsche Identität untergeschoben werden können. Andernfalls sind Angriffe möglich, die verheerend für die Sicherheit von RPs und Benutzern sind.

Privatsphäre spielt für die meisten SSO-Systeme leider keine Rolle. IdPs können ihre Benutzer aussphähen, da sie (per Design) erfahren, bei welchem RP sich ein Benutzer anmeldet. Dadurch können IdPs umfangreiche Profile erstellen und einige Benutzer werden von der Verwendung von SSO abgeschreckt. Zudem können IdPs ad-hoc entscheiden, ob sie einem Benutzer erlauben, sich an einem bestimmten RP anzumelden. Für den Schutz der Privatsphähre darf ein IdP nicht sehen können bei welcher RP sich ein Benutzer anmeldet. Bisher gibt es nur ein nennenswertes SSO-System mit diesem Ziel: Mozillas BrowserID (auch bekannt als Mozilla Persona).

Wir verwenden das Web Infrastructure Model (WIM) um die Sicherheit von SSO-Protokollen zu analysieren. Das WIM ist das bisher umfassendste formale Modell der Web-Infrastruktur. Wir erweitern das WIM, um auch Privatsphähreneigenschaften analysieren zu können. Basierend auf dem so erweiterten Modell führen wir die erste systematische und umfassende Analyse im Hinblick auf Privatssphäre in Web SSO durch.

Mit unserem Ansatz analysieren wir BrowserID. Als Ergebnis dieser ersten rigorosen Analyse eines SSO-Systems in der Web-Infrastruktur finden wir schwerwiegende Angriffe. Diese Angriffe kompromittieren nicht nur die Sicherheit von BrowserID, sondern zeigen auch, dass BrowserID keine Privatsphäre bietet. Wir empfehlen Korrekturen für BrowserID und zeigen, dass BrowserID damit sicher ist. In Bezug auf Privatsphäre ist BrowserID leider irreparabel.

Inspiriert von BrowserIDs Ziel entwerfen wir das erste Privatsphäre-schützende Web-SSO-System, genannt SPRESSO (für Secure Privacy-REspecting Single Sign-On). SPRESSO ist leicht zu bedienen, dezentralisiert und nutzt ausschließlich native Web-Bestandteile. Wir entwickeln SPRESSO von Anfang im WIM und beweisen formal, dass SPRESSO starke Sicherheitseigenschaften erfüllt und die Privatsphäre seiner Nutzer schützt.

1. Introduction

The World Wide Web has become a vital part of our modern information society. Built on top of the Internet, the Web is a universal information and communication platform and enables people around the globe to interact with each other and to exchange information. Many important and security-critical applications are based on the Web, for example, social networks, commercial services, or e-government portals. Users share private data, enter contracts, and even make monetary transactions online.

When users perform such sensitive actions, one important aspect is to *authenticate* users in a secure way. This means that a user needs to prove her identity to the services she wants to use. Only after successful authentication of the user, a Web service should grant access to its services and the user's resources stored at this service. For example, a social network needs to ensure that a user successfully authenticates herself before the user gets access to her personal messages and is allowed to post content.

Traditionally, each service on its own authenticates each user, typically by prompting the user for her username and password. To this end, a Web service maintains its own password database to be able to verify the password provided by the user. Secure password storage and management are non-trivial and hard to implement securely. Best practice recommendations (see, e.g., [Ope18]) are complex, extensive, and updated often. Many Web services realize their password management in a proprietary way leading to a huge set of different implementations in the wild. Such a very heterogeneous set of implementations and also responsibilities spread across many parties makes it hard to identify and fix (common) mistakes and security problems. Further, the services need to individually provide user support regarding authentication, such as password reset procedures.

Not only for Web sites, but also for users, password management is non-trivial. Users have to memorize many different passwords and are confronted with many different authentication dialogs. A password entered in the wrong place and sent to a malicious party easily leads to a compromised account. Likewise, if a user uses the same or similar passwords at different Web sites, a malicious Web site can easily guess this user's password at other Web sites.

Hence, it is very desirable to relieve users and Web services from the many downsides of this traditional approach. This is where *Web Single Sign-On (Web SSO)* comes into play. Web SSO

promises to relieve users and Web services from the burden of handling passwords.

1.1. Web Single Sign-On

In Web Single Sign-On (SSO), a Web service, in this context called *relying party (RP)*, delegates authentication to a so-called *identity provider (IdP)*. The task of an IdP in Web SSO is to authenticate users and attest their identities to RPs. Typically, some service that users use on a daily basis, such as email providers or social networks, serve as IdPs.

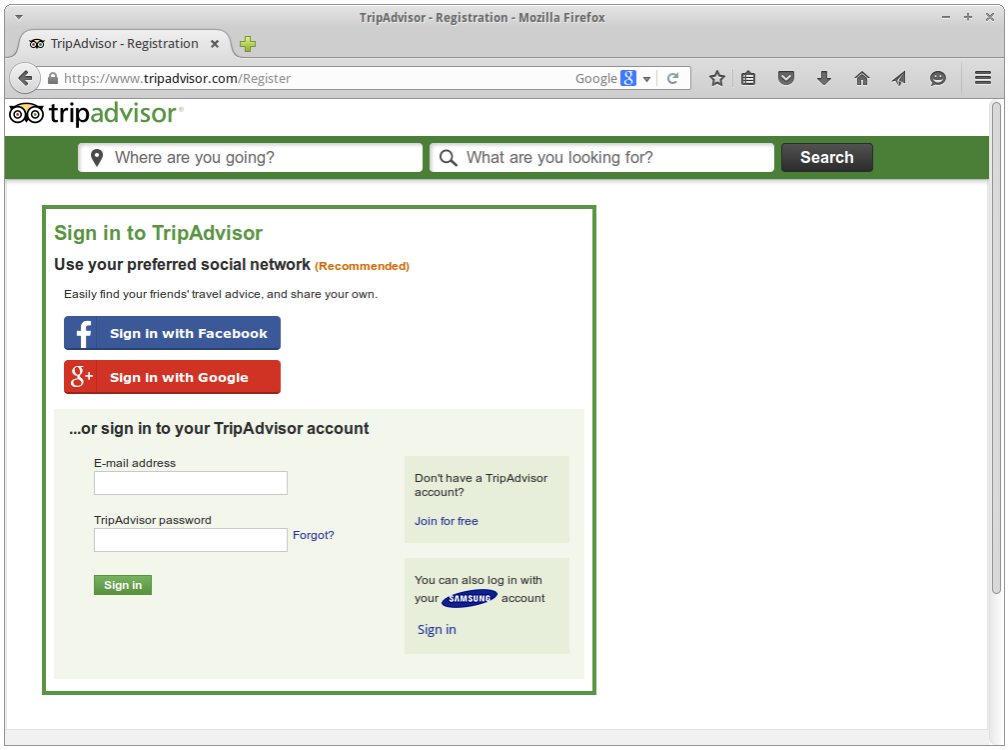
An SSO protocol defines the interaction between RP, IdP, and user. Many different SSO protocols have been developed so far. For the Web, the most important protocols include OAuth 1.0a [RFC5849] (used, for example, by Twitter), OAuth 2.0 [RFC6749] (used, e.g., by Facebook), OpenID 2.0 [FR+07] (used, e.g., by Yahoo), OpenID Connect 1.0 [Sak+14] (used, e.g., by Google and Microsoft), and Security Assertion Markup Language 2.0 (SAML) [Rag+08] (used, e.g., by Amazon AWS). A main feature of these modern Web SSO systems is that users are not required to have a special setup on their devices and can use standard Web browsers out of the box to log in. This sets these systems apart from other approaches such as Kerberos [RFC4120] or TLS client authentication [RFC8446].[1]

Web SSO provides many advantages for all parties. A user only needs to remember her credentials for one account (her account at the IdP) to log in at many different parties. If the user is already logged in at the IdP, the IdP might enable her to log in at many RPs without further interaction. Logging in at an RP where the user did not login before also becomes easier as the user does not need to register any authentication data at this RP. Using the SSO system, the user experience is always the same in all cases: the user only needs to interact with a familiar user interface (also reducing the user's susceptibility for phishing attacks). An example of a login flow from a user's perspective is given in Figure 1.1.

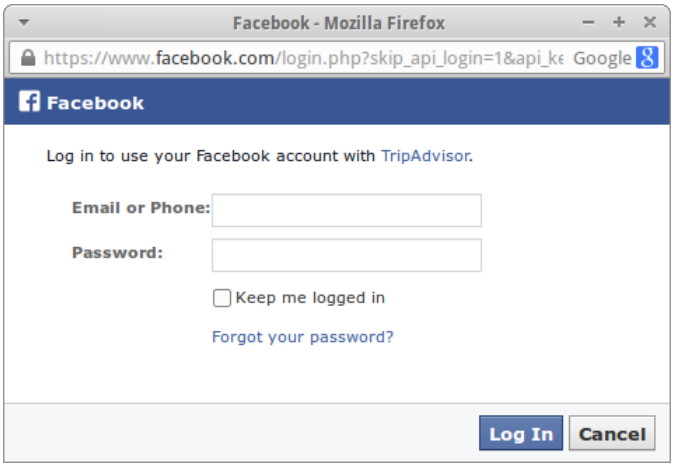
An SSO system allows the RP to outsource almost all authentication related tasks to a (set of) IdP(s). Hence, RPs do not need to develop and set up user registration and management systems and also do not need to handle user support concerning authentication (e.g., reset lost passwords). RPs are relieved from storing and protecting user credentials, which, as already mentioned, is not a trivial task.

While an SSO system seems to only bring high responsibility and cost to IdPs at first, also entities acting as IdPs can benefit from providing an SSO service: The IdP service provides added value for their users and therefore makes their services more appealing to new users.

[1] Note that while many browsers support TLS client authentication, users still need to create a new or import an existing certificate into the browser and cannot choose this login method ad-hoc. Besides this limitation, browser vendors have neglected the usability of this authentication method and provide sub-par user interfaces only.



(a) TripAdvisor login page.



(b) Facebook login dialog.

Figure 1.1.: A typical example of single sign-on in the Web: The travel information platform TripAdvisor provides several login options to users **(a)**. While a user can create a local account at TripAdvisor using her email address and a password, the user can also choose to use single sign-on using her Facebook or Google account (based on the protocol OAuth 2.0). TripAdvisor is the RP and Facebook or Google serve as the IdP. If the user, for instance, clicks on the button "Sign-in with Facebook", she is prompted to log in into Facebook using her Facebook account **(b)**. After successful authentication to Facebook, the user is then also logged in into the TripAdvisor Web site. When the user is already logged in at Facebook, the user can authenticate to TripAdvisor without further interaction. As the SSO protocols used in this example do not provide privacy towards the IdP, Facebook can easily observe that this user logs in at TripAdvisor.

1.2. Security in Web SSO

In the light of numerous attacks on SSO systems (see, e.g., [Arm+13; Arm+08; Ban+14; Cer+08; Gro03; HSN05; IET09; KR00; Som+12; SB12; SHB12; WCW12; YHR04; ZE14]), it is reasonable to put any SSO protocol under rigorous scrutiny. Such attacks can originate from many different sides, such as malicious RPs, third parties, or even IdPs. Clearly, security is crucial for an SSO protocol, and Web SSO systems obviously need to safeguard their users' identities.

An SSO protocol must ensure correct *authentication*, i.e., that nobody except the user herself should be authenticated under her identity to an RP. This means in particular that an adversary should be unable to authenticate himself such that an RP believes that this entity (the adversary) is some (different) user, i.e., the attacker cannot "break in" into the user's account at RP. Otherwise, an adversary can easily access the user's data or impersonate the user at the RP.

Another important, but not so obvious aspect is the *integrity of the user's login session*. That is, that an attacker should also not be able to force a user to log in at some RP or to manipulate a login process such that the user is logged in under some different account. The outcome of such attacks would be similar to the attack class of session swapping: The user might then interact with the RP under the attacker's identity and, for example, store confidential data at the RP that is then accessible under the attacker's account at RP.

Of course, such security goals can only be met based on certain assumptions: For example, the IdP of the user needs to be honest and attest the user's identity only if the user presents the correct password. If the adversary learns a user's password, he can easily impersonate himself to the IdP as the (honest) user and subsequently also log into any RP using the SSO protocol. This also requires that at least some communication must be performed via secure channels, e.g., when passwords are exchanged.

1.3. Privacy in Web SSO

The deployment of SSO might also impact privacy: The traditional approach of authentication in the Web, i.e., every RP authenticates its users on its own, does not involve any third party. Of course, entities running the network infrastructure can see that the user is communicating with the RP as well as other third parties that are meshed into the Web service itself (e.g., advertisements). The fact that a user is using a service of an RP is hidden from all other parties, e.g., some other Web services, that are not somehow involved (e.g., by hosting some parts of the RP's Web page). If SSO is introduced into this setting, then another third party, the IdP, which has not been involved before, is participating.

If the IdP can tell at which RP a user is logging in, this provides a huge impact on the user's

privacy. As authentication is centralized to the IdP, the IdP can easily create a detailed log containing every login at every RP the user is using. Such data can be used to create very extensive user profiles (e.g., for targeted advertisements) that do not only reveal the interests of a user, but might also allow to infer the user's sexual orientation, health issues, political opinions, a detailed schedule of the user's daily routines, or other personal information. People also tend to differently behave if they are aware of such surveillance (see, e.g., [BNR06]). A user might, for example, refrain from using SSO for authentication or from using some RPs at all. Following such ethical considerations, the legislation also requires companies to reduce the processing and storage of user data, for example, by the European Union's new general data protection regulations [GDPR].

Further, an IdP that knows exactly where a user is logging in, gains power over its users and RPs: The IdP can block certain RPs completely (e.g., to force RPs to pay for the SSO service or to enforce censorship) or ad-hoc decide whether it allows a user to log in at a certain RP, locking the user out from this RP and her data stored there.

In most Web SSO protocols, including the protocols mentioned in Section 1.1, IdPs always know exactly — by design — at which RPs their users log in. In Figure 1.1, for example, Facebook (the IdP) obviously knows that the user is about to log in at TripAdvisor, which is even displayed in Facebook's login dialog.

This problem would be solved by a privacy-preserving SSO system, i.e., a SSO protocol that hides the identity of the RP where a user is logging in from the IdP. So far, there has been only one Web SSO system claiming to provide privacy in the sense as defined above: BrowserID (a.k.a. Mozilla Persona). This SSO system has been developed by Mozilla to achieve this privacy goal explicitly. The question is whether this goal can be achieved with current Web technology, and in particular, if it is actually achieved by BrowserID. In this thesis, we will answer this question.

1.4. Formal Security and Privacy Analysis of Web Standards, Protocols, and Applications

Performing security and privacy analyses in the context of the Web is a non-trivial task. Since the initial proposal of the Web by Tim Berners-Lee in 1989 [Ber89], the Web has grown to a very complex infrastructure: Many different entities interact with each other using a variety of protocols and mechanisms. For example, if a user just opens a Web page by entering a URL, her browser triggers various actions, such as a DNS lookup, the establishment of a secure channel to this Web site's Web server using TLS, the request of the Web page itself using HTTP, the server

creating a response (which in turn can be based on a complex application-specific algorithm), processing of the response, rendering the actual Web page, executing scripts contained in the Web page, fetching additional resources, and so on.

The specification of these protocols and mechanisms is spread across many different standards and de-facto standards (such as proposals and common browser implementations). The standardization documents are managed by multiple organizations such as the Internet Engineering Task Force (IETF), the World Wide Web Consortium (W3C), and the Web Hypertext Application Technology Working Group (WHATWG). For example, the markup language for Web pages, Hypertext Markup Language (HTML), is developed by W3C [Fau+17] and WHATWG [WHA19b] in parallel. The retrieval of a Web page itself is defined in the Fetch standard by WHATWG [WHA19a], which makes use of the Hypertext Transfer Protocol (HTTP) protocol standardized by the IETF (with the very popular version 1.1 specified in [RFC7230; RFC7231; RFC7232; RFC7233; RFC7234; RFC7235]). Various other technologies, such as Cookies [RFC6265] or Strict Transport Security (STS) [RFC6797], are built on top of the data transfer via HTTP or complement it.

HTML documents can contain scripts written in JavaScript, a programming language developed for the Web [Ter18]. Such scripts can make use of various technologies, such as Web Messaging (postMessage) [Hic15], Web Storage (localStorage, sessionStorage) [Ian16], Cross-Origin Resource Sharing [Ann14], or XHRs [WHA19c] just to name a few.

This brief glimpse on important parts of the Web infrastructure illustrates that the Web is very complex. Moreover, many of the mechanisms mentioned above build upon other, complex technologies, such as the Internet and its transport protocols (e.g., TCP/IP [RFC791; RFC793] or TLS [RFC8446]).

Another aspect that adds complexity in today's Web is that many applications are not designed as standalone, isolated systems, but are interconnected or meshed together, i.e., they are composed of services provided by several different parties.

Unfortunately, the Web is a potentially hostile environment with a large group of participants having malicious intent. The inherent complexity of the Web plays into the hands of such adversaries. Developers can easily overlook problems that lead to attacks (see, e.g., [Ban+13; Ban+14; Han+10; Kar+07; SS13; SKS10; Sto+17; SB12; WCW12; Wan+11; Zhe+15]).

So far, the typical approach to check whether a Web standard (or application) meets its security and privacy goals, is to have groups of experts examine the protocol closely. Typically, these experts check whether known attacks are applicable. These experts might also discuss the protocol in detail and try to find new attacks, which obviously depends on the creativity of these analysts. Often, they do not formulate or even consider clear attacker models and make many implicit assumptions. Clearly, the result of such an analysis does not prove the non-existence of

unknown attacks. Further, while these experts have some intuition about the properties they want to prove, they cannot state these properties in a precise fashion as they only use natural language to describe a more or less vague concept.

Formal methods provide a systematic way to perform comprehensive analyses in a concise, rigorous mathematical fashion. To this end, the analyst first creates a formal model which reflects all relevant aspects of the system to analyze (e.g., a protocol or an application). This model then allows the analyst to precisely state all properties of interest in the analysis. Such properties can be functional properties, i.e., properties that require that a system matches its functional goals, or they can reflect security goals or privacy goals of a system. The analyst then tries to create a proof in a mathematical sense that the model actually fulfills the stated properties. As a model is an abstract representation of the real world, the result of a proof in the model can be translated to the real world (taking the modeling into account). For example, a security proof in a formal model proves security for certain classes of attacks in reality.

If a formal proof cannot be established, then this means that the model violates the properties in some way. Such a failed proof attempt can yield information about which part of the model is flawed. For example, in the context of a security property, this means that an attack exists and the step in which the proof failed provides hints about how the attack can be constructed. Further, the information gathered from such attacks can then be used by experts to develop fixes. These fixes can then be incorporated into the formal model, and the process of proving the properties can start over until the properties are proven.

In this thesis, we build on this approach and use formal methods to analyze the security and privacy of Web SSO protocols. In the remainder of this section, we take a closer look at formal approaches for the analysis of protocols and, in particular, the Web infrastructure.

1.4.1. Protocol Analysis

There exist two main approaches for formal protocol analysis: *symbolic analysis* and *computational analysis*. Symbolic analysis (initially proposed by Dolev and Yao in [DY83]) is based on formal terms that can be assembled and derived based on an equational theory. In the initial proposal, parties send and receive terms (messages) over a network that is completely controlled by an attacker, i.e., the attacker is able to intercept any message and to spoof new messages. In the world of symbolic analyses, cryptographic primitives are typically assumed to be perfect. For example, if the attacker observes a term $\mathsf{enc}_\mathsf{s}(s,k)$, which represents the symmetric encryption of some message s using a key k, the attacker cannot derive s from this term without knowing k (and without having learned s from somewhere else).

In contrast, in the so-called *computational* analysis (see, e.g., [SS84; SSR88]), messages are

bitstrings and are processed by (probabilistic) Turing machines. In particular, the attacker is also modeled as a (probabilistic) Turing machine with limited runtime. Cryptographic primitives are typically included in the analysis and are not considered as black boxes (as in symbolic analyses). This approach uses a lower level of abstraction, making it more precise including a stronger attacker model.

The computational analysis of very complex systems, however, can be very hard. To capture such systems and their complexity, symbolic approaches are more suitable due to their higher level of abstraction that focuses on analyzing the logic and interaction of protocols. Furthermore, although being less precise regarding cryptography, symbolic analyses still yield useful results.

A very famous example of a symbolic analysis is Lowe's attack [Low95] on the Needham-Schroeder public-key authentication protocol [NS78]. This protocol only consists of three simple steps and was considered to be secure for 17 years until Lowe published his remarkably simple attack on this protocol. This simple example shows that it is not sufficient to just have many experts "take a close look" at a protocol, but that a protocol's security needs to be analyzed and proven using rigorous methods. Such a proof (based on symbolic analysis) was carried out by Lowe for the fix he proposed in [Low96].

In this thesis, we have chosen to use a symbolic approach, which is ideally suited to capture the complexity of the Web.

1.4.2. First Approaches

Systematic analysis capturing the inherent complexity of the Web and also the protocols built on top of the Web would be very valuable. As we have seen in the Needham-Schroeder example above, even the analysis of simple protocols is non-trivial. So far, formal methods in this context have only been considered in a few approaches. In this section, we give an overview of first approaches before we cover the WIM, the most comprehensive model of the Web infrastructure to date, in the next section.

Early work in this direction includes work by Kerschbaum [Ker07], in which a mitigation against Cross-Site Request Forgery (XSRF) is proposed and formally analyzed using a simple model expressed using Alloy, a finite-state model checker that is based on a first-order logic language [Jac02]. This model includes a very abstract model of browsers and Web pages and is tailored to the XSRF setting.

A first formal foundation for the Web infrastructure has been proposed by Akahwe, Barth, Lam, Mitchell, and Song in [Akh+10], also based on Alloy. This work, however, includes only a very simple model of the browser and is not very expressive. In particular, this approach can yield only counter-examples, i.e., attacks, in bounded runs of the model. This work considers

three types of attackers: the *Web attacker*, which runs a malicious Web site and participates in the system as any other party, the *gadget attacker*, which is a slightly extended variant of the Web attacker and which in addition can inject (limited kinds of) content into otherwise honest Web sites, and the *network attacker*, which not only is able to provide malicious Web sites, but is also able to intercept and spoof network messages, i.e., messages exchanged between honest parties. This model was successfully used to find attacks on Web standards and protocols in [Akh+10].

In [Kum12; Kum14; Pai+11], Kumar et al. propose another approach based on Alloy extended with BAN logic to analyze the security of Web protocols. This approach, too, includes only a very limited set of Web features. The tool was used to analyze the Web authorization protocol OAuth 2.0 [RFC6749] as well as the Security Assertion Markup Language (SAML) [Rag+08]. The models consider one or more participants of a protocol to be malicious but do not take external attackers into account.

In [Ban+13; Ban+14], Bansal, Bhargavan, Delignat-Lavaud, and Maffeis propose the WebSpi model based on ProVerif [Bla13]. ProVerif is an automated tool to analyze cryptographic protocols (based on a variant of the applied pi calculus [AF01]). WebSpi includes many features of the Web infrastructure, such as Web Storage [Ian16], and is able to derive attack traces automatically. The model considers a network attacker, which is able to read, modify, and spoof the content of unprotected network messages. Still, this approach does not yield strong security proofs due to various limitations: imposed by the tool ProVerif, the WebSpi model is of monotonic nature. For instance, cookies and localStorage cannot be deleted or modified, but only be added. Cross-window interactions (e.g., postMessages or cross-window navigation) in browsers are not captured as the browser model does not include any window structure. Also, requests can only contain a limited number of cookies. Still, WebSpi was successfully applied to find vulnerabilities and attacks in encrypted cloud-storage services and the Web authorization protocol OAuth 2.0.

Besides these approaches, there are other approaches to analyze Web standards, protocols, and applications formally. These approaches, however, only consider limited models of the Web infrastructure. Bai et al. analyze the BrowserID protocol in [Bai+13]. Their work focuses on the automated extraction of a model for ProVerif from a protocol implementation. Their analysis of BrowserID is not very detailed, and only two rather trivial attacks are identified. For example, some critical protocol messages that when sent unencrypted, can be replayed by the attacker. We will also discuss their work in more detail as well as other work related to BrowserID in Section 4.5.3.

Bohannon and Pierce propose a formal model of a Web browser core [BP10]. Börger et al. present an approach for the analysis of Web application frameworks, focusing on the server [BCG12]. In [Arm+13; Arm+08], Armando et al. perform analyses of the SSO protocols SAML

and OpenID 2.0 [FR+07] focusing on a high-level abstraction of these protocols.

1.4.3. The Web Infrastructure Model (WIM)

The Web Infrastructure Model (WIM) is the most comprehensive model of the Web infrastructure to date. The WIM includes many relevant aspects of the Web. Besides basic infrastructure, such as Web and DNS servers, the WIM includes a very detailed model of a Web browser. This browser model captures many Web features, such as the handling of DNS, HTTP, and HTTPS messages, a detailed structure of windows and documents, an abstract model of JavaScript, Web storage and cookies, Web messaging (postMessage) and asynchronous HTTP communication (XMLHTTPRequest), a rich set of HTTP headers, HTTP redirections as well as security policies for cross-window navigation and access.

The WIM provides a concise view of all these aspects and allows modeling of complex modern Web applications and protocols, including systems composed of interactions between several entities via browsers.

For our analyses carried out in this thesis, the WIM is the only model available that is expressive enough. For example, BrowserID makes heavy use of Web Messaging across (nested) browser windows, which cannot be captured in other models.

We note that the author of this thesis is also a co-author of the WIM, which was first presented in [FKS14a] and subsequently extended and improved in follow-up publications [FHK19; FKS15a; FKS15b; FKS16a; FKS17b]. The development of the WIM itself is covered in the thesis of Daniel Fett [Fet18] and not part of this thesis. For self-containment of this thesis, we present the WIM in Chapter 2. During the work on this thesis, the author was continuously involved in the extension and improvement of the WIM. Our analysis of the BrowserID SSO system (see contributions below) is, in fact, the first analysis that was carried out using the WIM and part of the first publication of the WIM [FKS14a].

1.5. Contributions of This Thesis

In this thesis, we, in a first step, define a generic template for Web SSO systems. This template is independent of a specific SSO protocol. Based on this generic definition, we formally state the security and privacy goals introduced above. These generic definitions of the Web SSO system and properties serve as a foundation for our subsequent analyses and make their results comparable.

In the second step, we provide a detailed formal analysis of the security and privacy of a complex SSO system, namely BrowserID. As mentioned above, BrowserID was proposed by

Mozilla in order to create the first Web SSO system with privacy guarantees. We show, using formal methods, that the system does not achieve its privacy goal. The problem we discover is subtle and highlights the importance of formal analysis, which rigorously takes all relevant details into account.

In the third step, we then take the goal pursued by BrowserID and show that privacy is, in fact, achievable in a Web-based SSO system. We propose SPRESSO (Secure, Privacy-REspecting Single Sign-On), the first SSO system providing strong privacy and security guarantees. For SPRESSO, we provide a full formal model and formal proofs for its privacy and security guarantees. SPRESSO, therefore, is the only Web SSO system that provides this kind of privacy and is also the first with a formal proof for this property. Unlike BrowserID, SPRESSO does not rely on a centralized service, which by itself can easily break privacy.

We discuss our contributions in more detail below.

1.5.1. Generic SSO Web Systems and Generic Properties for Security and Privacy

As mentioned, our first step is to extract general characteristics of Web SSO systems and provide a generic definition of such systems. This definition captures important (but generic) user-driven events such as the user starting a login flow or entering her password. When modeling a concrete SSO system, these definitions serve as a template that is then refined to create concrete definitions for the SSO system. This template simplifies the modeling and analysis and is also reusable for others to analyze SSO systems.

Based on this generic definition, we identify and specify properties for the security of SSO systems, namely a property for authentication, i.e., a property that captures that a user's account can only be accessed by its owner, and a property for session integrity, i.e., a property that captures the integrity of a login flow.

For this generic SSO template, we also formulate our privacy property, i.e., that an IdP cannot see at which RP a user is logging in. In particular, we require that the IdP cannot distinguish between two login flows at different RPs. For this property to be fulfilled, all information that the user's IdP can possibly get in each login flow has to look exactly the same to the IdP.

These generic security and privacy properties are universally applicable to all SSO systems following the generic schema above. This way, analyses of different Web SSO protocols can be easily compared as the results are based on the same properties.

1.5.2. Analysis of BrowserID

As the Web SSO system BrowserID claims to provide privacy to its users, BrowserID is a very interesting system to analyze. BrowserID is also very complex and makes use of many modern Web features, making it a good example to exercise the expressiveness of the WIM.

At first, BrowserID was envisioned to be integrated into Web browsers, but to ease adoption, the actual realization only builds upon native features of the modern Web, such as Web messaging and Web storage. To achieve this goal, the protocol implemented in practice deviates from the original specification [Adi+13]. In particular, the envisaged browser integration is implemented as a Web application hosted by Mozilla. As a result, BrowserID has become a very complex SSO protocol that, for example, takes more than 80 steps for a typical login flow.

The BrowserID implementation also extends the original BrowserID specification with a second mode, in which Mozilla serves as an IdP that is tightly integrated with the core the protocol's implementation. This so-called secondary or fallback mode actually constitutes a separate protocol.

Extraction of the BrowserID model. As pointed out above, the specification of BrowserID only provides a high-level idea of its real-world counterpart. To get a comprehensive overview of all steps of the protocol, we manually analyzed the code of BrowserID (approx. 47k lines of code, written in JavaScript). Based on this analysis, we create separate models for BrowserID's primary mode and secondary mode using the WIM, which covers all features needed to describe BrowserID.

Each model is based on the WIM, i.e., the communication model as well as generic components, such as browsers and attackers and contains (1) a definition for Mozilla's server which provides the Web application component of BrowserID mentioned above along with the scripts that cover its browser-side parts, (2) a definition for IdPs (server and scripts), and (3) a definition for RPs (again, server and scripts). For these models, we follow our generic definition for SSO systems mentioned above and refine all generic SSO events such that our generic properties for security and for privacy are usable in these models.

Attacks on Security of BrowserID. During modeling and while trying to prove the security properties (authentication and session integrity) for BrowserID, we found several severe attacks that break the security properties.

In the *identity injection attack*, a malicious IdP is able to force the user to sign in at RPs using an identity which is not owned by the user. The *login injection attack* allows any malicious party to foist a different identity on a user. The *identity forgery attack* allows an attacker to exploit an error in the identity bridge feature of BrowserID, effectively enabling the attacker to authenticate

himself to any RP as any Gmail or Yahoo user. In the *key cleanup failure attack*, an attacker can get hold of a user's private key although the user thinks that her key is removed from a shared device. Similarly, in the *cookie cleanup failure attack*, the attacker is able to link his key pair to a user's account due to incorrect usage of cookies.

Security proof for fixed BrowserID. For all of the attacks on security mentioned above, we propose fixes and show that these fixes are indeed sufficient using a formal proof. To this end, we incorporate our fixes into the BrowserID models and show based on these models that both modes of BrowserID indeed satisfy the security properties. This security proof is the first security proof carried out using the WIM and the most comprehensive formal analysis of a Web application at the time of the publication [FKS14a].

Attacks on Privacy of BrowserID. While trying to analyze BrowserID's privacy, we discovered a severe attack (and several variants of this attack) that completely breaks privacy. A malicious IdP is able to probe whether a user logs into a specific RP based on the window structure in the user's browser. As it turns out, this problem is not easy to fix and would require a major redesign of BrowserID.

Further, as BrowserID relies on a central Web application provided by Mozilla, this application and its operator need to be trusted ultimately. Mozilla is able to track all login activities of BrowserID's users in all cases. This, in particular, applies to the secondary mode in which Mozilla itself serves as an IdP.

1.5.3. SPRESSO

Based on the lessons learned from BrowserID, we design a new SSO system for the Web: The *Secure, Privacy-Respecting Single Sign-On System (SPRESSO)*. This SSO system aims to provide security, privacy, and true decentralization. Further, SPRESSO relies only on native Web features and does not require any add-ons or modifications to the Web infrastructure.

Design. To design SPRESSO, we take a unique approach: After the initial design draft, instead of coding a proof-of-concept prototype, we first opt for the creation of a formal model based on the WIM. For development, the WIM provides a concise view on the Web infrastructure and thus, eases design decisions. This approach also enables us to perform a rigorous security and privacy analysis right away (see below).

For SPRESSO we need to master several challenges: To create an SSO system, RP and IdP need to be able to exchange data in some way that provides integrity while hiding the identity of the RP from the IdP. Also, this SSO system should be completely decentralized without the need for a central authority. Moreover, the SSO system needs to be usable ad-hoc without any

setup in the users' browsers. With SPRESSO, we achieve all of these goals and create an easy to integrate and easy to use SSO system.

As a side-effect of privacy, RPs can even seamlessly integrate SPRESSO into their traditional login procedures, i.e., when a user starts a login, the RP can automatically invoke an SPRESSO flow without any negative privacy implications.

Security Proof. Based on our model of SPRESSO built using the WIM, we show that the SPRESSO protocol indeed satisfies our security properties. As we designed SPRESSO with lessons learned from the BrowserID analysis in mind, we designed a much simpler protocol. As a result, the security analysis is less complex, more straightforward, and easier to comprehend.

Privacy Proof. We proved privacy, one of the main goals of SPRESSO. To this end, we show, based on our model, that a malicious IdP cannot learn any information about the RP from the authentication procedure. To this end, we show that the IdP's view is always the same, independent of the RP chosen by the user.

Proof-of-Concept Implementation. After successful analysis of SPRESSO using the WIM, we create a proof-of-concept implementation of all SPRESSO components. This implementation demonstrates that SPRESSO is indeed easy to use and simple to deploy at RPs and IdPs.

1.5.4. Generic Properties of the WIM

Besides continuous improvements of the WIM, we also identified and proved generic properties of the WIM, such as that the TLS abstraction of the WIM actually provides confidentiality and integrity. These generic properties are used in the analyses presented in this thesis and can also serve as helper lemmas for further analyses.

1.6. Publications

The contributions presented in this thesis are based on five peer-reviewed publications that have been published in international security conferences. All of these publications are complemented by accompanying technical reports. While the author of this thesis substantially contributed to all of these publications, some content is not part of this thesis.

An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System (S&P 2014) [FKS14a; FKS14b]. This work covers the analysis of the so-called secondary mode of BrowserID (see Section 4). Further, the WIM was initially proposed in this publication, but as already pointed out above, the WIM itself is not part of this thesis.

Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web (ESORICS 2015) [FKS14c; FKS15a]. This work complements the previous one by analyzing the so-called primary mode of BrowserID (see Section 4).

SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web (CCS 2015) [FKS15b; FKS15c]. In this publication, we propose the SSO system SPRESSO (see Section 5).

A Comprehensive Formal Security Analysis of OAuth 2.0 (CCS 2016) [FKS16a; FKS16b] and The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines (CSF 2017) [FKS17a; FKS17b]. In these publications, we further improve the WIM and present refined security properties. Further, we analyze the Web authorization framework OAuth and the Web SSO system OpenID Connect. Both analyses are again covered in the thesis of Daniel Fett [Fet18] and are not part of this thesis.

Improvements on These Publications. The work presented in this thesis further improves on the publications discussed above as follows:

We extracted all privacy and security definitions to a more generalized form (see Chapter 3). These generic definitions allow for easier adaption of these properties to different SSO protocols while maintaining a superordinate notion of privacy and security across SSO protocols.

For session integrity, we here use a notion derived from our analyses of OAuth [FKS16a] and OpenID Connect [FKS17b] that — in contrast to the session integrity property used in [FKS14a; FKS15a; FKS15b] — also requires that a user must have actually expressed her wish to get logged in under one of her identities using a specific IdP.

While in the original proposal of SPRESSO, we relied on a specific feature of HTML5 to suppress the so-called Referer header in order to protect a user's privacy towards the IdP, we here propose a slightly different version of SPRESSO that relies on a recently added Web feature called *Referrer Policies*. This change streamlines the protocol by eliminating a few steps.

We further note that in this thesis, all models and analyses have been adapted to a recent version of the WIM.

1.7. Structure of This Thesis

We first, in Chapter 2, present an overview of the WIM (incorporating our improvements) which we use as a framework to construct formal models of SSO protocols. In Chapter 3, we show how we can formalize SSO protocols, in general, using the WIM. This chapter is complemented by generic security properties which we use for our analyses in the subsequent chapters. We present BrowserID in Chapter 4 along with the formal models of the primary and the secondary mode and the analysis of these modes. We give a description of our new SSO protocol SPRESSO,

discuss the design decisions for this protocol, and present our formal model and our formal analysis of SPRESSO in Chapter 5. We conclude in Chapter 6 and provide an outlook on future work. Further details, such as formal definitions and the full proofs can be found in the appendix.

2. The Web Infrastructure Model (WIM)

The formal security analyses presented in this thesis are based on the Web Infrastructure Model (WIM), a generic Dolev-Yao style model of the Web infrastructure. The WIM has first been published in [FKS14a] and has been extended in follow-up publications [Fet18; FHK19; FKS15a; FKS15b; FKS16a; FKS17b]. Here, we recall this model following the descriptions in these publications with more technical definitions given in Appendix A.

As already explained in the introduction, the WIM is designed independently of a specific Web application and closely mimics published (de-facto) standards and specifications for the Web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM defines a general communication model, and, based on it, Web systems consisting of Web browsers, DNS servers, and Web servers as well as Web and network attackers.

2.1. Communication Model

The communication model of the WIM is illustrated in Figure 2.1. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes 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

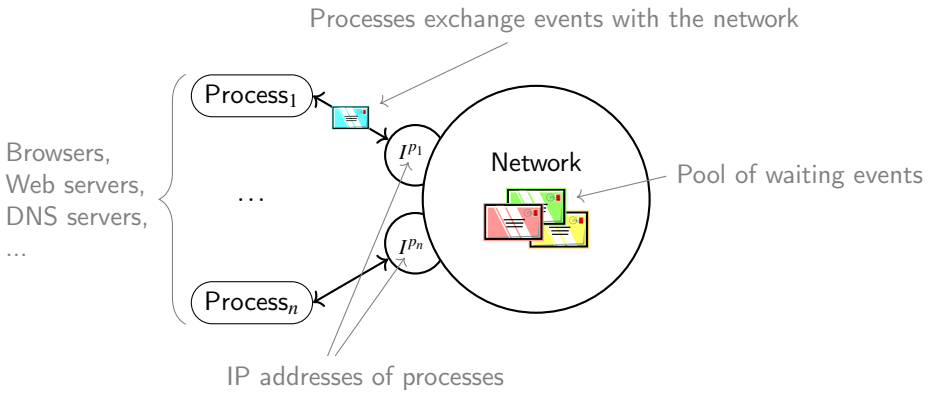


Figure 2.1.: Illustration of the WIM's communication model.

non-deterministically from a "pool" of waiting events and is delivered to one of the processes which listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

2.2. Terms, Messages, and Events

As usual in Dolev-Yao models (see, e.g., [AF01; DY83]), events and messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures).

Formally, 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 $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \Diamond\}$ 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: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, and $\mathsf{extractmsg}(\cdot)$,
- n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and
- projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

Based on Σ, we define terms as follows: Let $X = \{x_0, x_1, \dots\}$ be a set of variables and $\mathcal{N}$ be 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 $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models and is depicted in Figure 2.2. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k' \rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

$$\mathsf{dec_a}(\mathsf{enc_a}(x,\mathsf{pub}(y)),y) = x \tag{2.1}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x,y),y) = x \tag{2.2}$$
$$\mathsf{checksig}(\mathsf{sig}(x,y),\mathsf{pub}(y)) = \top \tag{2.3}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x,y)) = x \tag{2.4}$$
$$\pi_i(\langle x_1,\ldots,x_n\rangle) = x_i \text{ if } 1 \leq i \leq n \tag{2.5}$$
$$\pi_j(\langle x_1,\ldots,x_n\rangle) = \Diamond \text{ if } j \notin \{1,\ldots,n\} \tag{2.6}$$

Figure 2.2.: Equational theory for Σ.

For some term t, we use $t{\downarrow}$ to refer its normal form, i.e., a term t' with $t' \equiv t$ where all function symbols have been reduced from left to right as far as possible using the equational theory. For example, for some term $t = \mathsf{dec_a}(\mathsf{enc_a}(x,\mathsf{pub}(y)),y)$, we use $t{\downarrow}$ to refer to x.

For readability, strings (elements in $\mathbb{S}$) are depicted using a specific font. For example, `HTTPReq` and `HTTPResp` are strings. Further, we use a notation for *mappings* (dictionaries). For example:

$$[\texttt{dictkey1} : \texttt{value1}, \texttt{dictkey2} : \texttt{value2}] = \langle\langle \texttt{dictkey1}, \texttt{value1}\rangle, \langle \texttt{dictkey1}, \texttt{value1}\rangle\rangle$$

Full definitions of terms and notations can be found in Appendix A.1.

In the context of the Web, we define several specific sets of terms: We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of *domains*, e.g., $\texttt{example.com} \in \mathsf{Doms}$. By $\mathsf{Origins} \subseteq \mathsf{Doms} \times \{\texttt{P},\texttt{S}\}$, we denote the set of *(Web) origins* with the second element of the origin denoting the protocol, i.e., `P` denoting an (insecure) HTTP origin and `S` denoting a (secure) HTTPS origin of the domain (first element of the origin). For example, the term $\langle \texttt{example.com},\texttt{S}\rangle$ denotes the origin `https://example.com`. For HTTP(S) requests, we denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of *methods*, e.g., $\texttt{GET}, \texttt{POST} \in \mathsf{Methods}$.

As already mentioned above, entities in our model communicate via events that contain a message. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a,b,c\rangle,\mathsf{pub}(k))$ is interpreted to be the message $\langle a,b,c\rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

While messages can be arbitrary terms, we also define special kinds of messages:

- DNS messages

- HTTP messages,
- HTTPS messages,
- trigger messages, and
- corrupt messages.

While DNS and HTTP(S) messages model their real-world counterparts, *trigger messages* can be seen as "dummy messages" that are used to invoke actions in processes that are not taken in direct response to another message as in our communication model an action of some entity is always based on the processing of some message. For example, when a trigger message is delivered to a browser (which incorporates the behavior of a user, see below) and then this browser might (non-deterministically) follow some link on some Web page (currently opened in this browser). *Corrupt messages* are sent by attackers to honest parties in order to corrupt these parties, i.e., an honest party becomes a (collaborator of an) attacker. We will discuss corruption in more depth in Section 2.4 below.

DNS and HTTP(S) messages can be further divided into *requests* and *responses*. A response is associated with a request. To match a response to a request, both kinds of messages contain a nonce.[1] When a request is created by some process, this process freshly chooses this nonce. This nonce is then also used for the corresponding response message. For example, an HTTP request is represented as a term r containing the nonce mentioned above (say, n_1), an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP GET request for the URI `http://example.com/show?p=1` is represented as

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

where the body and the list of request headers are empty. A corresponding HTTP response is represented as

$$resp := \langle \texttt{HTTPResp}, n_1, 200, \langle\rangle, body\rangle$$

where the status code of the response `200` indicates that the request was accepted and processed by the server, the list of headers is empty, and *body* is a term containing the requested Web page.

For HTTPS messages, the underlying TLS channel is modeled in an abstract way as follows: The sender of the request (say, A, typically a browser) chooses a fresh symmetric key k' (a nonce) and includes k' in the request message. The request message is then asymmetrically encrypted

[1] The nonce of an HTTP(S) message models the TCP/TLS connection of the real world. DNS messages are typically sent as UDP messages in the real world and contain such a nonce by specification.

with the public key of the receiver (say, B). Such an HTTPS request for the HTTP request *req* above is of the form

$$\mathsf{enc_a}(\langle req, k' \rangle, \mathsf{pub}(k_{\text{example.com}})).$$

If $k_{\text{example.com}}$ (the private key for `example.com`) is only known to B, only B can decrypt this request message.[2] The symmetric key k' (now only known by A and B) is used by B to encrypt the response message, which can then later be decrypted by A using k'. Such an HTTPS response sent from B to A is of the form

$$\mathsf{enc_s}(resp, k').$$

An *event* (over IPs and $\mathcal{M}$) is of the form $\langle a, f, m \rangle$ for $a, f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f to be the sender address of the event. We denote by $\mathcal{E}$ the set of all events. Events can be compared to IP messages in practice, that carry some payload (a message) between two entities which are referred to by IP addresses.

For all formal definitions of messages and data formats, we refer the reader to Appendix A.2.

2.3. 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. We typically require that the events and the state that an atomic process outputs can be computed (more formally, derived) from the current input event and state. Such atomic processes are called *atomic Dolev-Yao processes (or simply, a DY process)*.

An atomic Dolev-Yao process $p = (I^p, Z^p, R^p, s_0^p)$ is defined to have

- a set of associated (IP) addresses I^p,
- a set of (possible) states Z^p ($\subset \mathcal{T}_N$),
- a process relation R^p that defines transitions from an (input) event and a (current) state to a set of (output) events and a (new) state such that both its output is derivable from its input,[3] and
- an initial state s_0^p ($\in Z^p$).

[2] In analyses, we typically show that a private key for a domain that is not controlled by an attacker is only known to its legitimate Web server (B in this example), i.e., the key is initially only known to this server and does not leak to any other party.

[3] We typically describe a process relation using pseudo-code. See Algorithm A.12 on Page 165 for a simple example.

We combine processes to a *system* $\mathcal{P}$ which is a (possibly infinite) set of atomic processes. A system itself does include state (except for the initial states of the processes). Also, as explained in the communication model above, processes communicate via events. Further, processes may use fresh nonces that have not been used before. We capture these aspects in a *configuration of a system*. A configuration is a tuple (S,E,N), which contains

- a mapping S from each atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$,
- an infinite sequence of waiting events E, i.e., $E = (e_1, e_2, \dots)$ where e_i are events that are about to be delivered to a process (for $i \in \mathbb{N}$), and
- an infinite sequence of (fresh) nonces $N = (n_1, n_2, \dots)$.

The sequence of waiting events E contains all events that have been sent by some process (but have not been delivered yet) and a (possibly infinite) number of trigger messages addressed to each (IP) address (interleaved by addresses). The sequence of nonces N is used to provide fresh, unique nonces to what we call a *processing step*. A processing step describes a transition from one configuration to a new configuration. In such a processing step, an event is non-deterministically taken from the sequence of waiting events, a process that is associated with the receiver's (IP) address of the event is selected,[4] and a new set of events and a new state for this process is (non-deterministically) derived using this process' relation. The relation might use nonces but does not assign them immediately. The nonces are described by placeholders, which are replaced with fresh nonces from the sequence of nonces by the processing step. We write, for example,

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

to denote the processing step from the configuration (S,E,N) to the configuration (S',E',N') in which some event e_{in} was delivered to some process p and p has outputted the set of events E_{out} (with S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces, i.e., N' contains all nonces from N except for the ones that are "used" in this processing step). We may omit the superscript and/or subscript of the arrow.

The output configuration of the processing step then contains

- the states of all processes S' (as a mapping as above), which are the same as in the previous configuration for each process except for the selected process,

[4] If multiple processes are associated with the same (IP) address, one of these processes is selected non-deterministically.

- the sequence of waiting events E' where the delivered event has been removed and events output by the process are added, and
- the sequence of nonces N' without nonces that are used in this processing step.

A *run* ρ of a system is a (possibly infinite) sequence of configurations, such that there exists a processing step between each consecutive configurations.

For readability, given a finite run $\rho = ((S^0,E^0,N^0),\ldots,(S^n,E^n,N^n))$ or an infinite run $\rho = ((S^0,E^0,N^0),\ldots)$, we denote by Q_i a processing step $(S^i,E^i,N^i) \to (S^{i+1},E^{i+1},N^{i+1})$ in ρ (with $i \geq 0$ and, for finite runs, $i < n$).

Based on this generic Dolev-Yao style model, we define *Web systems*. A Web system formalizes the Web infrastructure and Web applications. It contains a system $\mathcal{W}$ consisting of honest and attacker processes. Honest processes can be Web browsers, Web servers, or DNS servers. Attackers and (generic) honest processes are described in the next sections below. A Web system further contains a set of so-called scripts, which we will cover in Section 2.5. For each Web system, we also define the mapping addr that describes the ownership of (IP) addresses to DY process and the mapping dom that describes the ownership of domains to DY processes.

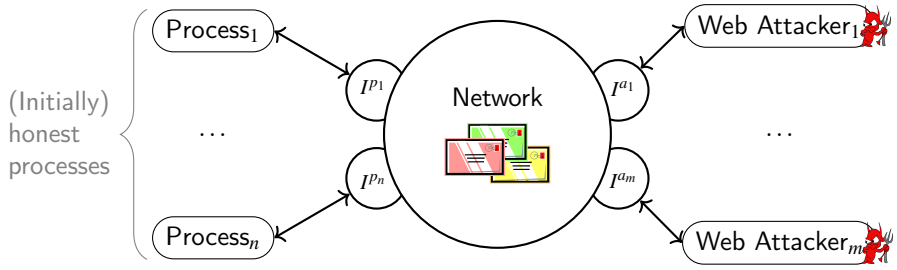
For full definitions of processes, systems, and runs, we refer the reader to Appendices A.3, for Web systems to Appendix A.8.

2.4. Attackers

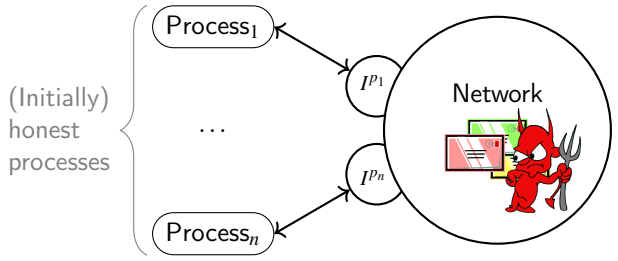
Attackers are modeled as processes in the WIM. An attacker is a Dolev-Yao process, which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. We distinguish two types of attackers: *Web attackers* and *network attackers*. Both types are illustrated in Figure 2.3. Web attackers participate in the network as any other process, i.e., they can receive events that are addressed to the respective Web attacker process, and they can send arbitrary events.[5] A network attacker[6] essentially controls the network. The network attacker can not only do the same actions as a Web attacker but can also receive messages that are not addressed to it. As the delivery of events is non-deterministic, all cases of a network attacker intercepting or blocking a message are captured.[7]

As already mentioned above, attackers can also corrupt other processes, e.g., browsers. If an honest process receives a corrupt message, it effectively turns into a Web attacker process

[5] The WIM allows any process to send arbitrary events, which includes IP spoofing.
[6] Note that one network attacker can subsume any number of network attackers and Web attackers.
[7] Note that an analysis in the WIM typically reasons about all possible runs of a system.



(a) Web attackers participate in the same way as any other process.



(b) A network attacker controls the network and can intercept and spoof events.

Figure 2.3.: Illustration of different attackers in the WIM.

that is given the process' state including secret values, such as credentials (passwords), cookies, etc.[8] Note that in security analyses, we typically require that certain processes do not become corrupted, while all other parties might be dishonest.

For the formal definition of attacker processes, we refer the reader to Appendix A.5. A mechanism for malicious Web pages opened in a browser is covered below.

2.5. Browsers and Scripts

In the WIM, a browser is a specific kind of DY process that mimics the behavior of a real-world Web browser. In Appendix A.6, we provide the full definition of a browser's state and relation.

A browser's state includes (among others) a list of open windows, cookies, and Web storage (localStorage and sessionStorage). A *window* inside a browser's state contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window (see Figure 2.4 for an illustration). Each document represents one loaded Web page and, again, includes a list of windows, creating a tree of windows and documents.

[8] In our security analyses of SSO protocols, an attacker initially does not have any credentials for any account at (honest) IdPs. Using corruption, the attacker can gain access to such credentials, modeling that the attacker can use accounts of (now dishonest) users at an IdP.

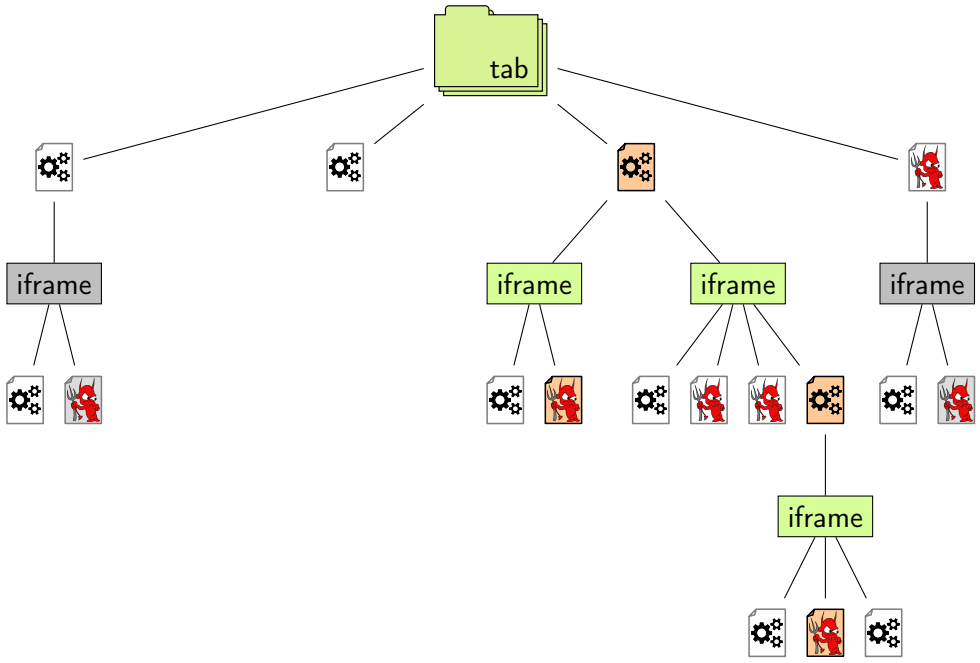


Figure 2.4.: Illustration of the window and document structure inside a browser. The depicted tree shows an example of the structure of one browser tab. Active documents are highlighted in orange.

Each document contains a *script*[9] and a state of this script (*scriptstate*). A script is a relation that models the behavior of one Web page and subsumes all components of this Web page including all external resources, such as external JavaScript files. Each document and its script behave similar to processes: Whenever a browser (during a processing step) decides to run a script in a document, the browser provides the script with its scriptstate (stored in the document) and a limited view on the browser's state (based on the document)[10] and expects to receive a command in return along with an updated scriptstate. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and Web storage data, and create iframes. Scripts can be either honest, in which case they model the behavior of the modeled application, or scripts can be dishonest, in which case they can derive any possible output, i.e., any output that is derivable from the script's input. We subsume all dishonest scripts in the so-called *attacker script* (R^{att}). Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant Web standards. See Section 2.5.2 below for more details on scripts.

One browser is thought to be used by one user, who is modeled as part of the browser. User

[9] More precisely, a document contains a script identifier which refers to a script relation.

[10] The view of a script includes a (limited) view on other documents and windows, certain cookies, Web storage data, and certain user credentials.

actions, such as following a link, are modeled as non-deterministic actions of the Web browser and scripts. User credentials are stored in the initial state of the browser annotated by an origin and are given to scripts according to the origin under which the script is running.

The browser relation takes care of processing many different types of messages, i.e., trigger messages, DNS responses, HTTP(S) responses, and corruption messages. As a result, the relation can then output messages such as DNS and HTTP(S) requests (including XHRs). When handling HTTP(S) messages, the relation takes care of important HTTP(S) headers, for example, cookie, location, strict transport security (STS), and origin headers.

Before we describe the handling of each of these messages in detail, we will first give a description of the dispatching mechanism for HTTP(S) requests, which is one of the browser's core functionality and spread across multiple processing steps.

2.5.1. Dispatchment of HTTP(S) Requests

If a browser wants to send out some HTTP(S) request (either as a result of a script, a URL bar action, or a reload), the browser prepares the (term of the) HTTP(S) request. Before sending this HTTP(S) request out, the browser first performs a DNS lookup for the domain to which the request is addressed to (recall that in the network model, all messages need to be addressed to some (IP) address, which is not a domain). For this DNS lookup, the browser creates an event that is addressed to some DNS server and that contains a DNS request message for the domain. The DNS server is determined by the browser's state, i.e., the browser is configured to send DNS requests to some (IP) address. The browser assigns the DNS request a fresh nonce as its message id. In the browser's state (namely in a subterm called *pendingDNS*), the browser records that it expects a DNS response that bears this message id. Further, along with this message id, the browser stores the HTTP(S) request that is to be sent out as well as a term *reference* that will be used to process the HTTP(S) response to this HTTP(S) request. The reference can be of two different kinds: (1) If the HTTP(S) request is an XHR (caused by a script), then the reference contains an identifier for the script's document as well as some term determined by the script (the script will later use this term to match the response to the request), or (2) in any other case, the reference contains an identifier for the window that will consume the HTTP(S) response, i.e., that will load the response's body as a new document. Finally, the browser outputs the event containing the DNS request and its modified state.

When the browser concludes the DNS lookup, i.e., when the browser receives a matching DNS response, it sends out the actual HTTP(S) request. At the same time, the browser moves the corresponding entry in the browser's state from the subterm *pendingDNS* to a subterm called *pendingRequests*. Entries in this subterm are used to match the corresponding HTTP(S) response

when it is received by the browser in a later processing step (see also Section 2.5.3 below).

In the case of HTTPS messages, the browser further takes care of encrypting the request. To this end, the browser looks up the public key of the receiving domain and chooses a fresh nonce as the (symmetric) key for the response (see also our explanation of HTTPS in Section 2.2 above). The nonce is stored along with the *reference* explained above.

Finally, if the browser receives a matching HTTP(S) response, it removes the corresponding entry from *pendingRequests* and processes the content of the message (see Section 2.5.4 below for more details).

2.5.2. Handling of Trigger Messages

A browser, at any time, can receive a trigger message upon which the browser non-deterministically chooses one of the following actions:

- Running a script in some document,
- mimic a user entering a URL in the location bar,
- reload a window, or
- navigate a window forward or backward.

We will describe each of these action in more detail below.

Running a script in some document. The browser non-deterministically selects a window and runs the script of the document contained within this window. As already mentioned above, the script subsumes the behavior of (a) a real-world HTML document that contains JavaScript and (b) user interaction with this document. Similar to process relations, a script is also non-deterministic and can use nonces by providing placeholders that will be filled with nonces by the process transition. The script relation maps from

- a limited view on the browser's window structure based on the document,
- an identifier of the current document in that window structure,
- the script's current *scriptstate*, a term that is stored in the document and that is used to store data of the script between different runs of that script (i.e., to keep state),
- a sequence of *script inputs*, a term that is stored in the document and that is used to store XMLHttpRequest responses and postMessages for this document,
- a sequence of cookies for the document's domain (which are not marked as HTTPonly),

- a dictionary containing the localStorage of the document's origin within the browser's state,
- a dictionary containing the sessionStorage of the document's origin within the current window tree,
- a sequence of ids of the browser's user, and
- a sequence of secrets (passwords) of the document's origin within the browser's state.[11]

to

- a new scriptstate that will be stored in the document,
- a new set of cookies for the current domain,
- a new dictionary for the localStorage as above,
- a new dictionary for the sessionStorage as above, and
- a command.

Commands issued by a script can be:

- `HREF`: The script outputs a URL and a window reference. This command models the user clicking on a link or a JavaScript navigating some window.
- `IFRAME`: The script outputs a URL and a window reference. This command models the script creating an iframe in the indicated window.
- `FORM`: The script outputs a URL, an HTTP method (GET or POST), form data, and a window reference. This models a user (or the JavaScript itself) filling out a form and submitting this form.
- `SETSCRIPT`: The script outputs a window reference and a script identifier. This command models the script replacing the content of the document of the indicated window.
- `SETSCRIPTSTATE`: The script outputs a window reference and a (second) scriptstate. This command models the script changing variables of JavaScript running in the document of the indicated window.

[11] Note that we model a user who is at least cautious about at which origin she enters passwords.

- `XMLHTTPREQUEST`: The script outputs a URL, an HTTP method, a term for the body of the HTTP request, and some term that will be used as a reference to match the response to this request at a later point in time. The response to an XMLHttpRequest will be later added to the script inputs of the current document.

- `BACK`, `FORWARD`, or `CLOSE`: The script outputs a window reference. This command models the script navigating the indicated window backward or forward, or closing the window. The navigation over the history of a window changes a flag that marks the currently active document in the window's history.

- `POSTMESSAGE`: The script outputs a window reference, a message term, and an origin. This command models the script sending a postMessage to the indicated window. (The origin is used to restrict the delivery of this message to documents of this origin.) The postMessage will be added to the active document's script inputs in the indicated window.

A browser immediately processes the command output by the script (within the same processing step). For most commands, certain restrictions apply. For example, a script may only instruct the browser to replace a script by the SETSCRIPT command if the target document is of the same origin as the current document. A browser ignores invalid or forbidden commands.

Some commands, however, require interaction with the network. The browser relation then prepares the HTTP request, outputs the corresponding DNS request and stores the HTTP request along with some other metadata (e.g., which document/window, XMLHttpRequest reference, ...) to later continue processing the command when the DNS response is delivered.

Mimic a user entering a URL in the location bar. The browser non-deterministically decides whether it creates a new (top-level) window. If it does not create a new window, it non-deterministically selects some existing top-level window. The browser also non-deterministically chooses a URL (that does not contain any nonces and hence no secrets). For the selected (new or existing) window, the browser behaves as an (imaginary) script of this window (more precisely of the document within that window, assuming such a document exists) issued the `HREF` command for the chosen URL and a reference for this window.

Reload a window. The browser non-deterministically selects some window and navigates this window to its current document's URL similar to the `HREF` command as described above. This choice models a user's click on the reload button in her browser's navigation bar or of the context menu of some window (which might be an iframe).

Navigate a window forward or backward. The browser non-deterministically selects some window and then behaves similar to the `BACK` and `FORWARD` commands as described above. This

mimics a user's click on the backward or forward button in her browser's navigation bar or the context menu of some window (which again might be an iframe).

2.5.3. Handling of DNS Responses

When the browser receives an event containing a DNS response, it looks up in its state whether it expects such a response. If the browser does not find an entry in *pendingDNS*, it ignores the DNS response. If the browser finds an entry, the browser considers the DNS lookup to be finished and removes this entry from *pendingDNS*. Now the actual HTTP request will be sent out. The browser checks whether the HTTP request is to be sent to an HTTPS URL, i.e., the browser determines whether the original message needs to be encrypted. As mentioned above, the browser chooses a fresh nonce that will be used as a symmetric key to encrypt the HTTP response later and encrypts the HTTP request along with this nonce asymmetrically with the public key of the domain (taken from the browser's state). The browser now takes this encrypted term[12] or the plain HTTP request (depending on whether the URL is HTTPS or not) as the actual message and creates a new event with this message that is addressed to the (IP) address contained in the DNS response. Further, the browser creates a new entry in its state in the subterm *pendingRequests*. This entry essentially contains the same information as the former entry in *pendingDNS* (HTTP request, reference) plus the (IP) address and (in the case of HTTPS) the nonce that will be used as the symmetric key to encrypt the response. Finally, the browser outputs the event containing the HTTP request as well as its modified state.

2.5.4. Handling of HTTP(S) Responses

When the browser receives an event containing an HTTP(S) response, it uses the subterm *pendingRequests* in its state to look up whether it expects such a response. If it does not expect such an event, the browser ignores it. Otherwise, the browser finds an entry in this subterm that contains all necessary information to process this response (including a symmetric key to decrypt the response in the case of HTTPS). Similar to DNS responses, the browser removes this entry from *pendingRequests*. The browser now checks whether the HTTP(S) response contains a redirect (i.e., whether the response contains a redirect status code) and continues as described below.

The HTTP response does not contain a redirect. As above, when the browser created the corresponding request, the browser again has to distinguish two cases:

[12] Note that we only consider symbolic encryption in the WIM.

- The response is for an XMLHttpRequest. In this case, the browser uses the reference term contained in the entry in *pendingRequests* to retrieve the document's identifier and the script's reference term that was used when a script instructed the browser to send out the request. The browser looks up the document in its window structure and appends the script's reference along with the body and selected headers of the HTTP response to the list of script inputs. The script inputs can be used by the script running within this document when it is triggered in a later processing step.
- The response is for any other kind of request. In this case, the browser expects the HTTP response's body to contain a script identifier and an initial state for this script. The browser creates a new document with this information. This document is then appended to the history of the window identified by the window identifier contained in the entry in *pendingRequests*. This document is also marked as being the active document of this window.

The HTTP response contains a redirect. In this case, the browser follows the redirect except if the corresponding HTTP request was caused by an XMLHttpRequest (the browser knows this from the reference term of the entry that it took from *pendingRequests*). Following a redirect means that the browser behaves similar to sending the HTTP request in the first place: The browser creates an event with a DNS request for the domain of the URL it is redirected to and creates an entry in *pendingDNS* that uses the same reference term as for the first HTTP request (this term is contained in the entry the browser took from *pendingRequests*. Depending on the type of the redirect (the WIM implements the redirect codes 303 and 307, which subsume the behavior of all kinds of redirect codes), the request's body is dropped.[13]

2.5.5. Corruption

A browser can also become corrupted as explained above. We model two types of corruption of browsers, namely *full corruption* and *close-corruption*, both of which are triggered by special network messages in the WIM. In the real world, an attacker can exploit buffer overflows in Web browsers, compromise operating systems (e.g., using trojan horses), and physically take control over shared internet terminals.

Full corruption models an attacker that gained full control over a Web browser and its user. Besides modeling a compromised system, full corruption can also serve as a vehicle for the attacker

[13] If the HTTP request was a POST request, the request might contain request parameters in its body. In the case of a 307 redirect, the new request remains the same as the old request except for the URL it is directed to and the *Origin* HTTP header. In the case of a 303 redirect, the request is changed to a GET request, and the body is removed.

to participate in a protocol using secrets of honest browsers (think of the attacker "recruiting" collaborators).

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

2.6. Web Servers

Web servers are application-specific processes. The primary function of a Web server is to respond to HTTP(S) requests (with HTTP(S) responses). A Web server might also send out HTTP requests on its own (e.g., invoked by a trigger message) and process responses to its requests.

As Web servers model application-specific behavior, they need to be defined depending on the respective application. To ease the definition of the behavior of Web servers, the WIM provides a generic server template. This template provides basic functionality to handle incoming HTTPS requests as well as sending out HTTPS requests. To model application logic in this framework, several "dummy" functions can be replaced with application specific behavior (i.e., pseudo code).[14]

Note that a Web server can typically also become corrupted as explained above.

2.7. DNS Servers

DNS servers respond to DNS requests, which ask for the IP address of some domain. The DNS server looks up this IP address and responds with a DNS response. 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. Hence, DNS servers contain a list of assignments of domain names to IP addresses in their state and use only their state to look up the result of a DNS query, i.e., they do not perform further DNS resolving by forwarding the DNS request to another DNS server. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover specific attacks on the DNS system itself.

Here, we only model plain DNS and do not consider recent improvements and extensions such as DNSSEC [RFC4033] and DNS over HTTPS [HM18], which provide integrity (in the case of

[14] As we do not make use of the generic server template in this thesis, we do not provide a detailed description and refer the reader to [Fet18].

DNSSEC) and confidentiality (in the case of DNS over HTTPS). This is motivated by the fact that these technologies are not mandatory and — in the worst case — not used at all. Hence, omitting these techniques is a safe overapproximation. In a system that contains a network attacker, we typically do not consider honest DNS servers but configure all parties to use the attacker as their DNS server. In such a system, the attacker is always able to intercept and respond to all DNS messages and hence, we do not get any guarantees from an honest DNS server. DNS servers might also become corrupted, giving Web attackers also the ability to interfere with DNS resolution.

2.8. Limitations

Of course, a model cannot reflect all aspects of the real world as a model is always an abstraction. As such, the WIM also abstracts away some aspects: There are no details of programming languages, such as JavaScript, or byte-level attacks, such as buffer overflows. Still, the WIM can capture the outcome of such attacks by using dynamic corruption or the attacker script within the browser. The WIM also omits user interface details, which might miss user-level attacks such as Clickjacking. Further, the WIM, as a Dolev-Yao style model, uses an abstract view on cryptography and TLS. Still, the WIM provides us with a comprehensive view on the logic of interactions between different entities, or even scripts within a browser, and allows analyses to yield meaningful results for this abstraction level.

2.9. General Security Properties of the WIM

We have identified central application independent security properties of web features in the WIM and formalized them in a general way such that they can be used in and facilitate future analysis efforts of web standards and web applications. Roughly speaking, these properties show that our model for HTTPS is sane. In our analyses, we often rely on these properties. In this section, we provide a brief overview of these properties, with precise formulations and proofs presented in Appendix B.

The first set of properties concerns encrypted connections (HTTPS): We show that HTTP requests that were encrypted by an honest browser for an honest receiver cannot be read or altered by the attacker (or any other party), given that the attacker does not know the private TLS key of the receiver. This, in particular, implies correct behavior on the browser's side, i.e., that browsers that are not fully corrupted never leak a symmetric key used for an HTTPS connection to any other party. We also show that honest browsers set the host header in their

requests correctly, i.e., the header reflects an actual domain name of the receiver, and that only the designated receiver can successfully respond to HTTPS requests.

The second set of properties concerns origins and origin headers. Using the properties stated above, we show that browsers cannot be fooled about the origin of an (HTTPS) document in their state: If the origin of a document in the browser's state is a secure origin (HTTPS), then the document was actually sent by that origin. Moreover, for requests which contain an origin header with a secure origin, we prove that such requests were actually initiated by a script that was sent by that origin to the browser. In other words, in this case, the origin header works as expected.

3. Formalizing SSO Protocols and Security Properties

In this chapter, we present a generic template to formalize SSO protocols in the WIM. This template is a template for a Web system which we then call an *SSO Web system*. The template includes DY processes for relying parties and identity providers which will be refined for analyses of concrete SSO protocols. Browsers, DNS servers, and attackers are as defined in the previous chapter.

For this template, we also specify certain important processing steps that are common to all SSO protocols. These processing steps represent user actions, such as the start of a login flow, as well as the success of an SSO login. We further introduce the notion of an SSO session that captures every processing step related to an SSO login. This notion allows us to reason about instances of SSO protocol flows.

Using these definitions, we define security properties that every SSO protocol should fulfill. As already explained in the introduction, these properties cover authentication, integrity of the user's login session (which we call session integrity), and privacy.

3.1. SSO Web Systems

The goal of an SSO protocol is to authenticate a user u to an RP r using some account a at some IdP i. As we consider open and federated SSO protocols, in which many parties can participate, the account a needs to have some globally unique identifier id. In practice, such an identifier is typically composed of some domain $domain$ of i and some local identifier of the user $name$, i.e., id is very similar to an email address.[1]

Note that the identifier id can also be some URL, which is, for example, specified for some configurations of OpenID 2.0 [FR+07]. Still, such a variant of the identifier can be split into a domain and a part local to this domain. There exist, however, schemes like *extensible resource*

[1] Note that the domain $domain$ does not necessarily "belong" to the IdP i, i.e., the domain may be assigned to some different entity, which in turn, may delegate the governance over id to the IdP i. Such a delegation can also be implicitly set by the protocol, e.g., in the case of BrowserID's secondary mode in which a trusted entitiy takes the role of an IdP for all accounts.

identifiers (XRI) [LLW05] that have been developed to enable users to have an identifier that is independent from a specific IdP or a domain.[2]

Some SSO protocols, such as OpenID 2.0, also allow one account to be identifiable by multiple (different) identifiers at the same time, e.g., a URL, some string similar to an email address, and an XRI identifier. In the case of OpenID 2.0, this is handled by a lookup protocol in which all identifiers of one account are normalized to a single unique identifier. Without such a normalization, different identifiers would be treated as identifiers for different accounts from the SSO protocol's perspective.

In this thesis, we focus on modeling identifiers similar to email addresses as described above. Note that this design choice does not restrict our approach. Our template can be easily extended to support more complex identifier schemes.

Definition 1 (User Identities). A *user identity* (or identity or id for short) *id* is a term of the form $\langle name, domain \rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$.

Let ID be the finite set of identities and y a Dolev-Yao process. By ID^y we denote the set $\{\langle name, domain \rangle \in \mathsf{ID} \mid domain \in \mathsf{dom}(y)\}$. ◇

We say that an identity *id* is *governed* by some DY process i (taking the role of an IdP). Formally, we define a (protocol-specific) mapping $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$. As explained above, the governor i of some identity *id* is typically the DY process to which the domain of the identity belongs, i.e., $\mathsf{governor} : \langle name, domain \rangle \mapsto \mathsf{dom}^{-1}(domain)$. Note that the governor of some identity might also be malicious, i.e., we also consider dishonest IdPs.

To authenticate herself as the rightful owner of an identity *id*, a user needs to authenticate to the governor of *id*. To perform the actual authentication of a user, there are a variety of different authentication schemes: The user can just be asked by the IdP i to provide some credential (e.g., a password). In other schemes, a user might perform a multi-factor authentication, e.g., by entering a (long-term) password and some one-time password that she received via a different channel (e.g., SMS). We consider the actual authentication scheme that is used to authenticate a user to the IdP to be outside of the SSO protocol itself. We therefore resort to the widely used password-based approach that is solely based on the knowledge of the password and does not incorporate any additional security measures. Note that this design decision does not restrict us to extend our analyses to include stronger authentication schemes. In our formalization, we define a bijective mapping from identities to passwords (i.e., *secrets* which are modelled as nonces)

[2] In XRI, users register their identifier at a central registry, which keeps record of which IdP is currently responsible for a specific identifier. In this case, the identifier itself does not contain a domain, but lookup of this identifier at the registry returns a domain of the IdP.

that assigns secrets to all identities as the authentication credential. We denote this mapping by $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Secrets}$.

Recall that users in the WIM are modelled as part of browsers. To assign identities to users, i.e., browser processes, we assign secrets to browsers: Let $\mathsf{ownerOfSecret} : \mathsf{Secrets} \to \mathsf{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret. As a shortcut, we further define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$, $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which maps identities to browsers (owning the respective identities). We typically distribute secrets to owners (browsers) and governors (IdPs) of the respective identities using the initial states of the processes.

RPs are DY processes that conclude a protocol flow by considering some user (identity) to be authenticated. This final step is characterized by the RP to issue some nonce n, which we refer to as a *service token*.[3] The service token refers to a (now) authenticated *service session* between the RP and and the user (browser). In practice, n can be used as a session cookie and the RP stores n along with the *id* of the authenticated user in its state.[4]

For our template, we define RPs as follows:

Definition 2 (Relying Parties, Service Tokens, and Service Sessions). A *relying party* (RP) $r = (I^r, Z^r, R^r, s_0^r)$ is a Web server to which a user is authenticated as the goal of (a flow of) an SSO protocol where r meets the following requirements:

1. When r concludes its part in a run of the SSO protocol, i.e., the RP considers the authentication of some user under some id *id* to be completed, r issues a freshly chosen nonce n, which we call a *service token*.

2. There exists a function $\mathsf{serviceSessions} : Z^r \to [\mathcal{N} \times \mathsf{ID}]$ which extracts a dictionary over service tokens and identities from the RP's state where every service token n (issued by r and considered valid in the respective state of r) is mapped to the identity *id* for which n was issued. We call $\langle n, id \rangle$ a *service session*.[5]

[3] We require that an RP needs to choose a fresh nonce to create an authenticated Web session (e.g., using a cookie) with the user at the end of the protocol, which is a best practice for authentication (see, e.g., OWASP session management recommendations [Ope17]).

[4] In alternative approaches, RPs might track service sessions outside of their state. For example, the RP can create a token that contains (among others) a MAC over the user identity and store this in a cookie in the user's browser. The exact mechanism how RPs implement service sessions is, however, outside the actual SSO protocol.

[5] We typically store this dictionary as a subterm in RP's state, i.e., the function $\mathsf{serviceSessions}$ is a projection from RP's state in this case. Note that, in practice, an RP might also store additional information related to an authenticated session. Such information might be stored as additional information in the same dictionary, but could also be stored in a different subterm in RP's state, e.g., in a dictionary that contains such information indexed by n.

3. For every processing step

$$Q = (S,E,N) \xrightarrow[r \to \{e_{\text{out}}\}]{} (S',E',N')$$

in which r concludes a flow of the SSO protocol for the id id and issues the service token n (contained in e_{out}), we have that

$$\mathsf{serviceSessions}(S'(r)) = \mathsf{serviceSessions}(S(r)) +^{\langle\rangle} \langle n, id \rangle.$$

For every processing step

$$Q' = (S'',E'',N'') \to (S''',E''',N''')$$

in which r does not conclude any flow of the SSO protocol, we have that

$$\mathsf{serviceSessions}(S'''(r)) = \mathsf{serviceSessions}(S''(r)).$$

4. Initially, an RP is honest. When an RP processes a corrupt event, the RP becomes corrupted and behaves as a Web attacker from then on.

◇

As we can see, this template for RP does not contain any SSO protocol specific parts. For a concrete SSO protocol, we refine this definition to capture all protocol specific parts. Similar, IdPs are defined in a very generic way. The main requirement for IdPs is the role that they play in the SSO protocol:

Definition 3 (Identity Providers). An *identity provider* (IdP) is a Web server that is able to attest a user's identity to an RP in an SSO protocol. Similar to RPs, an IdP is initially honest. When an IdP processes a corrupt event, the IdP becomes corrupted and behaves as a Web attacker from then on. ◇

We combine all these definitions to a generic template for an SSO Web system:

Definition 4 (SSO Web System). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be a Web system. We call $\mathcal{WS}$ an *SSO Web system*, iff the set $\mathcal{W}$ can be partitioned into $\mathcal{W} = \mathsf{Hon} \dot\cup \mathsf{Web} \dot\cup \mathsf{Net}$ with Web attacker processes in Web, network attacker processes in Net, and a finite set of (initially) honest parties Hon, which can be further partitioned into $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP} \cup \mathsf{DNS} \cup \mathsf{Other}$ with 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, a finite set Other of other (honest) parties. ◇

For the first two security properties below (authentication and session integrity), we consider SSO Web systems that contain a network attacker and no Web attackers (i.e., $\mathsf{Web} = \emptyset$ and $\mathsf{Net} = \{\mathsf{attacker}\}$). For privacy, we consider SSO Web systems with a set of Web attackers.

Note that a network attacker can not only observe and spoof events exchanged among other processes, but also has the power to subsume multiple other network or Web attackers. Also note that in a SSO Web system with a network attacker, we can safely overapproximate such a system by omitting DNS servers as all interactions with an honest DNS server can be intercepted and simulated by a network attacker.

3.2. Important Steps in an SSO Flow

As mentioned in the introduction, SSO protocols are intended for delegated authentication: A user can log into an RP by authenticating herself at an IdP. In such a flow, we can identify four distinctive steps: First, the user tells the RP that she wants to start the SSO flow. Second, the user states which IdP she intends to use. Third, the user states the identity that she wants to use. Finally, the user gets logged in at the RP. Note that some of these steps may happen in the very same step: Figure 3.1 shows an example of an RP where the user is prompted to log by selecting an IdP out of a predefined list. In this example, the user starts the SSO login flow and selects the IdP for that login flow at the same time. As we will see in our analysis of BrowserID, the selection of the IdP and the selection of the identity is performed in the very same step.

We will now provide generic definitions of all of these steps as far as generality allows. In concrete SSO protocols, these steps will be tied to specific elements of process relations (i.e., lines of code). These definitions allow us to precisely identify the steps in our session integrity property below.

We first define the start of the SSO protocol flow as follows:

Definition 5 (Start of an SSO Flow). Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be a processing step, b a browser, and r an RP. We write $\mathsf{started}(Q, b, r)$ iff in Q, the browser b started an SSO flow at r. ◇

Next, we define two choices of the user, i.e., the selection of the IdP a user wishes to use and the selection of an identity a user wishes to use:

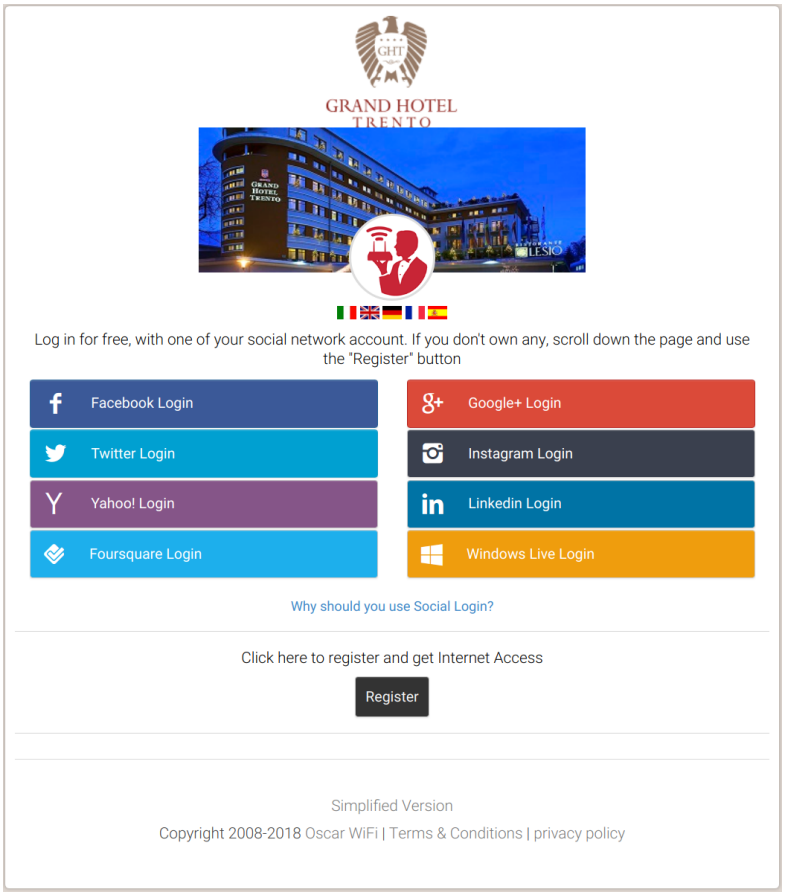


Figure 3.1.: A screenshot of a typical RP Web page (the Wifi login portal of the Grand Hotel Trento) that prompts the user to select one out of many different IdPs to log in.

Definition 6 (Selection of an IdP). Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be a processing step, b a browser, and i an IdP. We write $\mathsf{selectedIdP}(Q,b,i)$ iff in Q, the browser b stated to use the IdP i to log in. ◇

Definition 7 (Selection of an Identity). Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be a processing step, b a browser, i an IdP, and id an identity. We write $\mathsf{selectedID}(Q,b,i,id)$ iff in Q, the browser b selected to use the identity id (using the IdP i)[6] to log in. ◇

Finally, we define the conclusion of an SSO protocol flow to be the processing step in which the user gets logged in at an RP:

[6]Note that the IdP i is typically given by the selection of the identity id implicitly.

Definition 8 (User is Logged in at RP). Let $\mathcal{WS}$ be an SSO Web system. Let ρ be a run of $\mathcal{WS}$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, id an identity, and n some term. We write $\mathsf{loggedIn}(Q, b, r, id, n)$ iff in Q, b receives n as a service token (i.e., b interprets n to be a service token, e.g., by storing n as a cookie) and n identifies a service session in the state of r (in Q) and this service session is authenticated for the identity id. ◇

3.3. Trace Sessions and SSO Sessions

A *trace session* (in some run) is a set of processing steps that relate to each other in some way. We will define a notion of trace sessions based on causal chains of processing steps. Intuitively, we say that two processing steps are connected if one processing step causes the other. This can happen either based on network events (i.e., one DY process handles an event output by another process) or based on a browser's state (i.e., the script of the some document is invoked that has been loaded or modified in the earlier processing step or the script of the same document is invoked in both processing steps or the script of the document in the later processing step has been influenced by a script of the earlier processing step). Formally, our definiton of connected processing steps is as follows:

Definition 9 (Connected Processing Steps). Let ρ be an run of some Web system and $x, y \in \mathbb{N}$ with $x < y$. Further let

$$\begin{aligned} Q_x &= (S^x, E^x, N^x) \xrightarrow[p_x \to E_{\text{out},x}]{e_{\text{in},x} \to p_x} (S^{x+1}, E^{x+1}, N^{x+1}) \text{ and} \\ Q_y &= (S^y, E^y, N^y) \xrightarrow[p_y \to E_{\text{out},y}]{e_{\text{in},y} \to p_y} (S^{y+1}, E^{y+1}, N^{y+1}) \end{aligned}$$

be processing steps in ρ. We say that Q_x and Q_y are *connected* iff

(1) $e_{\text{in},y} \in E_{\text{out},x}$, or

(2) $p_x = p_y$, p_y is a browser, $e_{\text{in},y}$ is a trigger event, the browser p_y (in Q_y) decided to run the script of document d_y (i.e., selected `script` in Line 9 of Algorithm A.9 and selected d_y in Line 13 of Algorithm A.9), and

   a) $e_{\text{in},x}$ is an HTTP(S) response message that either

      i. created d_y in Lines 39ff. of Algorithm A.8, or

      ii. modified the subterm *scriptinputs* of d_y in Lines 56ff. of Algorithm A.8, or

b) $e_{\text{in},x}$ is a trigger event, the browser p_x (in Q_x) decided to run the script of some document d_x (i.e., selected `script` in Line 9 of Algorithm A.9 and selected d_x in Line 13 of Algorithm A.9), and

   i. the document selected in Line 13 of Algorithm A.9 is the same in Q_x and Q_y, or

   ii. the browser p_x (in Q_x) modified (as a result of running the script of d_x) the document d_y, i.e., modified a subterm of d_y in the state of p_x.[7]

◇

Note that the definition above assumes that in the modeling of a concrete SSO protocol, honest RPs and IdPs do not use trigger events to continue a sequence of logically connected steps. The definition only captures trigger events handled by browsers to continue the execution of scripts. Obviously, if in some SSO protocol the usage of trigger events by non-browser parties is necessary to model this protocol, the definition of connected processing steps needs to be extended accordingly.

Based on the notion of connected processing steps, we now define *trace sessions* to be sequences of connected processing steps.

Definition 10 (Trace Sessions). A *trace session (in a run ρ of a Web system)* is a sequence of processing steps $(Q_0,\ldots,Q_n)$ or $(Q_0,Q_1,\ldots)$ such that (1) for all Q_i with $i>0$, Q_i is connected to some processing step in $(Q_0,\ldots,Q_{i-1})$, and (2) all processing steps appear in the same order as in ρ. ◇

Based on the notion of trace sessions, we now define *SSO sessions*. Intuitively, an SSO session starts when a user expresses her wish to log in at some RP. Each session can only contain one such request. A session ends when the log in is complete (which does not necessarily happen in all SSO sessions).

Definition 11 (SSO Sessions). Let $\mathcal{SSO}$ be an SSO Web system and ρ be a run of $\mathcal{SSO}$. An *SSO session in ρ by a browser b with an RP r* is an infinite trace session $(Q_0,Q_1,\ldots)$ or a finite trace session $(Q_0,\ldots,Q_n)$ in ρ such that $\mathsf{started}(Q_0,b,r)$, but there is no $j>0$ such that $\mathsf{started}(Q_j,b,r)$. If there exists $j>0$ such that $\mathsf{loggedIn}(Q_j,b,r,i,id)$ for some IdP i and some identity id, then the SSO Session is finite and $n=j$. ◇

We write $\mathsf{SSOSessions}(\rho,b,r)$ for the set of all SSO Sessions in ρ by b with the RP r.

[7] Such a modification is performed in Line 58, Line 62, or Line 95 of Algorithm A.7 as a result of the value of *command* output by the script (in Line 10 of Algorithm A.7) being of the form $\langle$`SETSCRIPT`, *window*, *script*$\rangle$, $\langle$`SETSCRIPTSTATE`, *window*, *scriptstate*$\rangle$, or $\langle$`POSTMESSAGE`, *window*, *message*, *origin*$\rangle$ with *window* pointing to the window (in the browser's state) where d_y is the active document of that window and *script*, *scriptstate*, *message*, *origin* being some terms.

3.4. Security Properties

Based on the generic definition of SSO Web systems, we now define security properties for such a system. As already discussed in the introduction, we identify three fundamental properties that can be informally described as follows:

Authentication 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 identity of an honest user (browser) governed by an honest IdP.

Session Integrity The attacker should not be able to impose a false identity on an honest browser, i.e., he should not be able to authenticate an honest browser to an honest RP with an identity that is not owned by the browser.

Privacy An IdP (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.

Formally, we define these properties as follows.

3.4.1. Authentication

For the authentication property, we require that an SSO system cannot be in any configuration in which the attacker can derive a service token that was assigned by an honest IdP for an identity of an honest browser that is not fully corrupted. Recall that full corruption of a browser means that an attacker gets access to all secrets owned by this browser. In a browser that is only close-corrupted, the attacker does not get to know these secrets because certain parts of the browser's state are cleared when it becomes corrupted (recall that this variant of corruption models shared internet terminals, e.g., an internet kiosk in a hotel lobby, where the user closes all windows before handing over the machine to the next person, which might be an malicious).

Definition 12 (Authentication Property). Let $\mathcal{WS}^n$ be an SSO system with a network attacker (denoted by $\mathsf{attacker}$). We say that $\mathcal{WS}^n$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{WS}^n$, every configuration (S,E,N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser $b \in \mathsf{B}$ that is not fully corrupted in S, every identity $id \in \mathsf{ID}$ owned by b with $\mathsf{governor}(id)$ being an honest IdP, every service session $\langle n, id \rangle$ at r, n is not derivable from the attacker's knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$). $\diamond$

3.4.2. Session Integrity

We now define a security property which we call *session integrity*. This property states that an attacker is not able to force an honest user to be logged in at an honest RP without the user giving her explicit consent that she indeed wants to be logged in at this RP. Further, this property requires that the user is actually logged in using an IdP that the user choose to log in and under one of her own identities that the user choose to log in. In other words: This security property captures that a user should only be logged in under one of her identities and that the user actually expressed the wish to start an SSO flow using a certain IdP and this identity. Formally, we require that the actual login and user's selections are connected to each other in a direct causality chain (i.e., they are part of a single SSO session) and that the created service session matches the user's selections.

Note that for this security property, when considered in a setting with network attackers, one cannot rely on traditional cookies to pass the service token to the browser, as cookies lack integrity in the presence of a network attacker. The network attacker can trivially inject a cookie into a browser over an insecure HTTP connection such that this cookie is also used for HTTPS connections [Zhe+15]. If one still wants to use cookies, there is a recent proposal to fix the integrity of cookies: If a cookie name is prefixed by the string `__Secure`, this cookie can not only be set in browser over secure connections. In addition, such a cookie can also be later identified as a cookie that was indeed set over a secure connection [BW17]. (Another way is to not rely on cookies at all and pass the service token in a different way, e.g., as some JavaScript variable.)

Definition 13 (Session Integrity). Let $\mathcal{SSO}$ an SSO Web system with a network attacker. We say that $\mathcal{SSO}$ is *secure w.r.t. session integrity* iff for every run ρ of $\mathcal{SSO}$, every processing step Q_{login} in ρ, for every browser b that is honest in Q_{login}, for every RP $r \in RP$ that is honest in Q_{login}, and for every $id \in IDs$ the following holds true: If there exists some n such that $\mathsf{loggedIn}(Q_{\text{login}}, b, r, id, n)$ we have that all following statements hold true:

i) $\exists o \in \mathsf{SSOSessions}(\rho, b, r)$,

ii) $\exists Q_{\text{IdP}} \in o$ such that $\mathsf{selectedIdP}(Q_{\text{IdP}}, b, r, \mathsf{governor}(id))$,

iii) $\exists Q_{\text{id}} \in o$ such that $\mathsf{selectedID}(Q, b, i, id)$,

iv) $Q_{\text{login}} \in o$,

v) b owns id.

◇

We note that for SSO systems in which the user selects the identity in a document controlled by the IdP, the session integrity property above cannot hold true if the IdP is dishonest. In the analyses of OAuth 2.0 and OpenID Connect 1.0 in [FKS16a; FKS17b], a slightly weaker session integrity property was used: There, the Statement iii) was only considered under the condition that $\mathsf{governor}(id)$ is honest. This integrity property is, however, still reasonable as the usual attacks on session integrity are targeted towards a user interacting with an RP under an identity she does not own. In this case, however, we still have that this identity is governed by some IdP, which in turn can always impersonate all identities of its users and access the services of any RP under such an identity.

We further note that the results of our BrowserID and SPRESSO analyses presented in [FKS14a; FKS15a; FKS15b], consider a weaker session integrity, namely only the condition stated in Statement v) above.

3.4.3. Privacy

So far, for the authentication and session integrity properties above, we considered a network attacker, which is the most powerful kind of attacker. As the privacy property presented below reasons about hiding the fact which RP the user is interacting with from the IdP, the property is stated in a slightly different setting that considers only Web attackers. A network attacker can trivially break such a privacy property by just observing the addresses of network events.

In the privacy property, we require only a very limited set of honest parties: two honest RPs r_1 and r_2 with the domains dr_1 and dr_2, respectively, one honest browser, one honest DNS server, and — depending on the SSO system — a limited set of other parties. For these parties, we also require that they cannot become corrupted, i.e., these parties never collaborate with an attacker. These honest parties interact with a very hostile environment: All other parties, including all IdPs, are dishonest and modeled as Web attackers. This implies that every login flow involves attackers.

Before we state the privacy property, we first introduce the notion of indistinguishability of Web systems. This notion is used to formulate privacy such that an attacker cannot distinguish between two Web systems in which a user logs into r_1 or r_2, respectively.

In the following, we provide high-level ideas and the most important definitions and refer the reader to Appendix C for the remaining formal definitions.

Indistinguishability of Web Systems. Indistinguishability of Web systems follows the idea of trace equivalence in Dolev-Yao models (see, e.g., [CCD11]), which in turn is an abstract version of cryptographic indistinguishability. The indistinguishability definition presented here is not tailored towards a specific Web application, and hence, is also of independent interest.

Intuitively, two Web systems are indistinguishable if the following holds true: whenever the attacker[8] 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 [AF01] (see below). More specifically, since, in general, Web systems allow for non-deterministic decisions (in particular, decisions taken by honest parties), the sequence of actions of the attacker might induce not a single run but a large set of (different) runs. Hence, we allow the attacker to "fix" all of these non-deterministic decisions, effectively allowing him to control also honest parties in a limited way.

This gives the attacker effectively more power in this case. Recall that previously, he only controlled corrupted parties and certain scripts running in honest browsers (most notably the attacker script R^{att}). Now, he might also control a user's actions in a browser. He can, for example, instruct the user to navigate any window of her browser.

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 a term of the form

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

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 $cmd_{\text{switch}} \in \{1,2,3\}$ 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, cmd_{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

[8] Note that there might also be multiple attackers in a system at the same time. For readability, we here describe this set of all attackers as one single entity.

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 $\mathcal{WS}$, a schedule σ induces a set of (finite) runs in the obvious way. We denote this set by $\sigma(\mathcal{WS})$. If all processes and scripts in $\mathcal{WS}$ are deterministic, only one run is induced.

Before we can define indistinguishability of two Web systems, we, as mentioned above, recall the definition of static equivalence by Abadi and Fournet [AF01]. We say that two terms 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

$$\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k)) \approx \mathsf{enc_a}(\langle r', k' \rangle, \mathsf{pub}(k)) \ .$$

In this example, if the attacker does not know k, this statement is true.

As Web systems may contain multiple attackers and as we base indistinguishability on one attacker's view, we need to specify one distinguished attacker process as the "origin" of this view. Note that this design does not restrict attackers, as all other dishonest processes can always send their view to this distinguished attacker process.

Definition 14 (Web System with Distinguished Attacker). Let $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be a Web system and p an attacker process in $\mathcal{W}$. We say that $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0, p)$ 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. We define *SSO Web systems with a distinguished attacker processes* analogously. ◇

We define indistinguishability of Web systems as follows:

Definition 15 (Indistinguishability). Let $\mathcal{WS}_0$ and $\mathcal{WS}_1$ be two Web systems each with a distinguished attacker process p_0 and p_1, respectively. 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 the two Web systems $\mathcal{WS}_0$ and $\mathcal{WS}_1$ *indistinguishable* iff they are indististinguishable under all schedules σ. ◇

Formal Definition of Privacy. In order to set up two Web systems that can be used to define privacy based on indistinguishability, we introduce a special kind of browser that we call a *challenge browser*. A challenge browser is a browser that is parameterized with a *challenger domain*. When this browser assembles an HTTP(S) request for the special "dummy" domain `CHALLENGE`, then instead of putting together and sending out the request for `CHALLENGE`, it takes the challenger domain. However, this is done only for the first request to `CHALLENGE`. Further requests to this domain are not altered (and would probably fail, as the domain `CHALLENGE` is presumably not listed in the honest DNS server). To this end, the state of the challenge browser contains a flag (i.e., the state is extended with one more subterm that contains this flag) that stores whether the challenge browser has replaced the placeholder `CHALLENGE` with the challenger domain.

We use the challenge browser to create two Web systems that are exactly the same up to the challenger domain. As we are interested in whether a dishonest IdP can distinguish if a user logs in into some (honest) RP A or into some (honest) RP B during a run, we set in one case the challenger domain to a domain of RP A and in the other case to a domain of RP B. For privacy, we require that in all possible runs induced by all schedules, the attacker cannot distinguish whether the challenge domain in the challenge browser is the one of RP A or RP B. If the attacker can distinguish between both cases, then the SSO protocol must have leaked this information and does not provide privacy.

To model privacy, we define an *SSO Web system for privacy analysis* that is a Web system with a distinguished attacker process $\mathsf{attacker}$. This Web system is parameterized by a (challenger) domain and contains one (honest) challenge browser b that uses this domain as the challenger domain.

Definition 16 (SSO Web System for Privacy Analysis). We call $\mathcal{WS}^{\text{priv}}(\cdot) = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *SSO Web system for privacy analysis* if it is an SSO Web system with a distinguished Web attacker $\mathsf{attacker}$ and

1. there is no network attacker in $\mathcal{W}$ (i.e., $\mathsf{Net} = \emptyset$ and $\mathsf{attacker} \in \mathsf{Web}$),
2. there exists at least one honest DNS server $\mathsf{dns} \in \mathcal{W}$,
3. there exist two honest RPs $\mathsf{r}_1, \mathsf{r}_2 \in \mathcal{W}$ with dr_1 a domain of r_1 and dr_2 a domain of r_2,
4. there exists exactly one challenge browser $\mathsf{b} \in \mathcal{W}$ that takes the parameter of $\mathcal{WS}^{\text{priv}}$ as its challenger domain and that is configured to only use the honest DNS server dns, and
5. dns, r_1, r_2, and b cannot become corrupted.

◇

As explained above, for privacy, we set up two SSO Web systems with parameter dr_1 and the same one with parameter dr_2. We require that the attacker's view in both systems for all runs that are induced by an arbitrary schedule[9] is statically equivalent, i.e., both systems are indistinguishable. Hence, the attacker then cannot decide which RP is used by the challenge browser, i.e., the attacker cannot distinguish between dr_1 and dr_2 as the challenger domain in b.

Definition 17 (Privacy Property). Let $\mathcal{WS}^{\text{priv}}(\cdot)$ be an SSO Web system for privacy analysis with dr_1 and dr_2 the domains of the two (honest) relying parties, b the challenge browser, and $\mathsf{attacker}$ the distinguished attacker process as defined above. Let

$$\mathcal{WS}_1 := \mathcal{WS}^{\text{priv}}(dr_1) = (\mathcal{W}_1, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker}_1) \text{ and}$$
$$\mathcal{WS}_2 := \mathcal{WS}^{\text{priv}}(dr_2) = (\mathcal{W}_2, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker}_2)$$

with $\mathsf{attacker}_1 = \mathsf{attacker}_2 := \mathsf{attacker}$, $\mathsf{b}_1 := \mathsf{b}(dr_1)$, $b_1 \in \mathcal{W}_1$, $\mathsf{b}_2 := b(dr_2)$, $b_2 \in \mathcal{W}_2$, 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 $\mathcal{WS}^{\text{priv}}$ is *IdP-private* iff $\mathcal{WS}_1$ and $\mathcal{WS}_2$ are indistinguishable. ◇

Note that there are many different situations where the honest browser in $\mathcal{WS}^{\text{priv}}$ 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 of privacy 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 dr_1 or dr_2, i.e., whether this resulted in a login request for dr_1 or dr_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 dr_1 or dr_2. Other requests in a run, even to both dr_1 and dr_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 dr_1 or dr_2.

[9] As explained above, the attacker can effectively choose which schedule is used as we require that the attacker's view needs to be statically equivalent for runs induced by all possible schedules.

4. Analysis of BrowserID

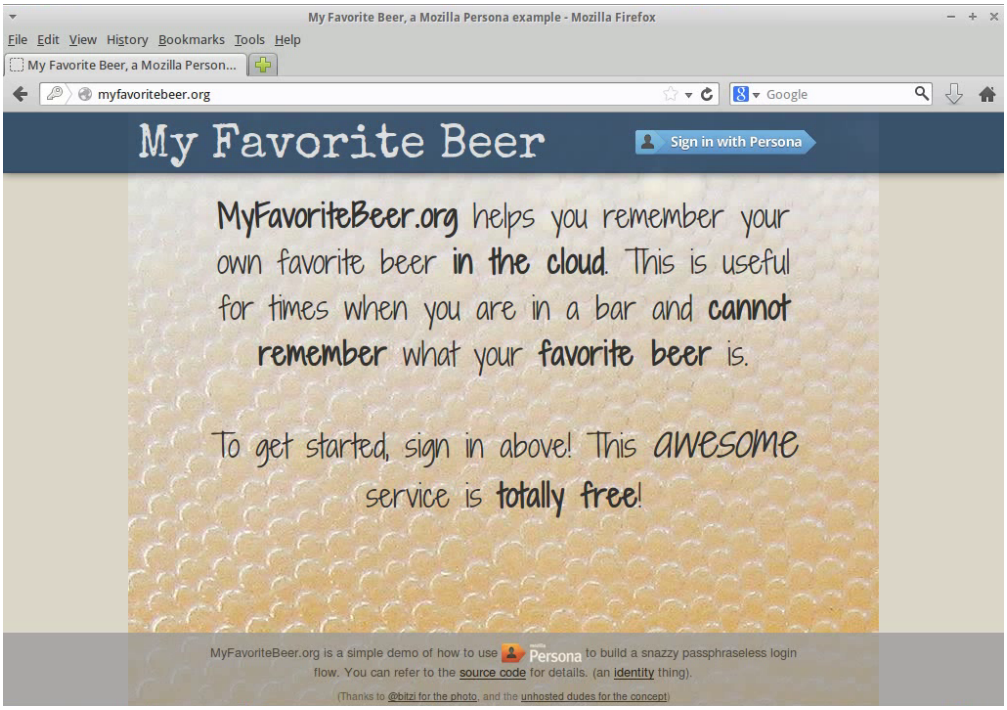
BrowserID [Adi+13] is a decentralized SSO system developed by Mozilla for user authentication on Web sites (BrowserID was marketed as *Mozilla Persona* [Moz18]).[1] Although decommissioned by Mozilla in November 2016 [Kel16], BrowserID is a very interesting SSO system as its distinctive promise is to respect users' privacy, i.e., identity providers should not be able to track where their users log in [Bam+18b].

User identities in BrowserID are email addresses and email providers are meant to become IdPs for BrowserID. To ease wide adoption by users, Mozilla also developed an extension to the initial BrowserID protocol that enables users without a BrowserID-enabled email provider to use this SSO system. In this extension, a central IdP run by Mozilla is automatically used for these identities (see *secondary mode* below).

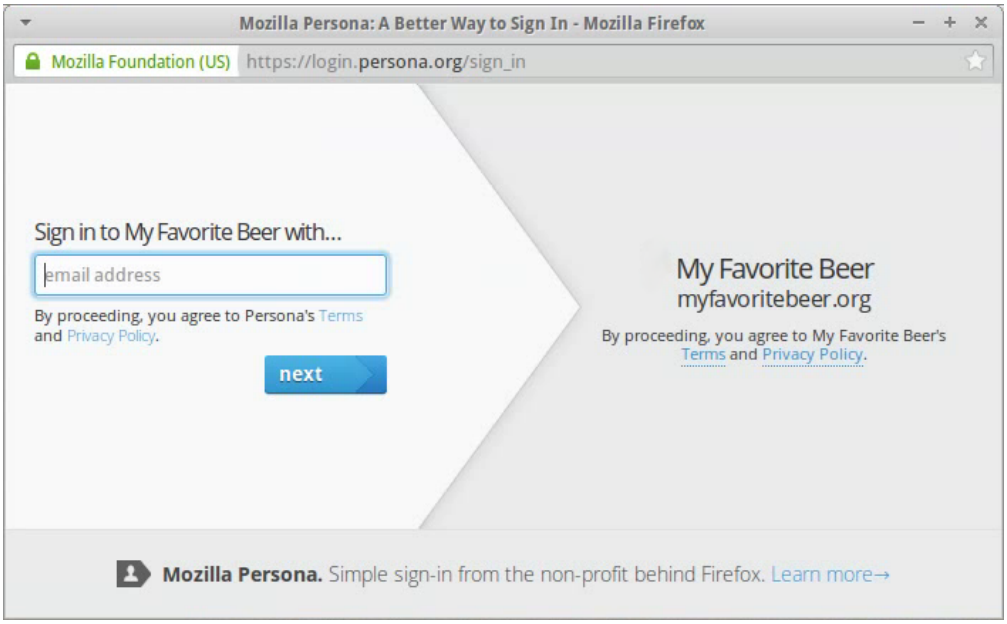
In Figure 4.1, we illustrate a user's perspective of BrowserID. The figure shows screenshots of the steps a user performs to log in at a demo RP provided by Mozilla. The user first starts a login (Figure 4.1a), enters her email address (Figure 4.1b) and her password (Figure 4.1c), and is finally logged in (Figure 4.1d).

At first, BrowserID was envisioned to be integrated into Web browsers, i.e., browsers provide a BrowserID API that provides necessary key management and communication features. To ease adoption, the developers opted for implementing BrowserID using native Web features: APIs needed for RPs and IdPs were provided as JavaScript libraries and the core of the protocol and its GUI as a Web application that was hosted by Mozilla (which we denote by LPO after its domain "login.persona.org"). This bundle provides the same functionality as the (envisioned) browser integration [Moz13] and makes use of a broad variety of browser features, such as XHRs, postMessage, Web Storage (local- and sessionStorage), cookies, etc. and consists of ~47k LOC (excluding code for Sideshow/BigTent, see below, and some libraries). This implementation was deployed in practice and used in, for example, the blogging software Wordpress [TB15] as well as Mozilla's own Web services, such as their bug tracker [Mar12]. As the browser integration was never realized, the implementation based on native Web features is the only existing one.

[1] Strictly speaking, the term Persona refers to Mozilla's implementation of the BrowserID protocol. We will, however, use the term BrowserID synonymously for both, the protocol and the implementation. This is motivated by the fact that Mozilla's implementation is the only implementation of the BrowserID protocol and describes many important parts of the interaction within the browser, which are not fixed in the original protocol specification [Adi+13].

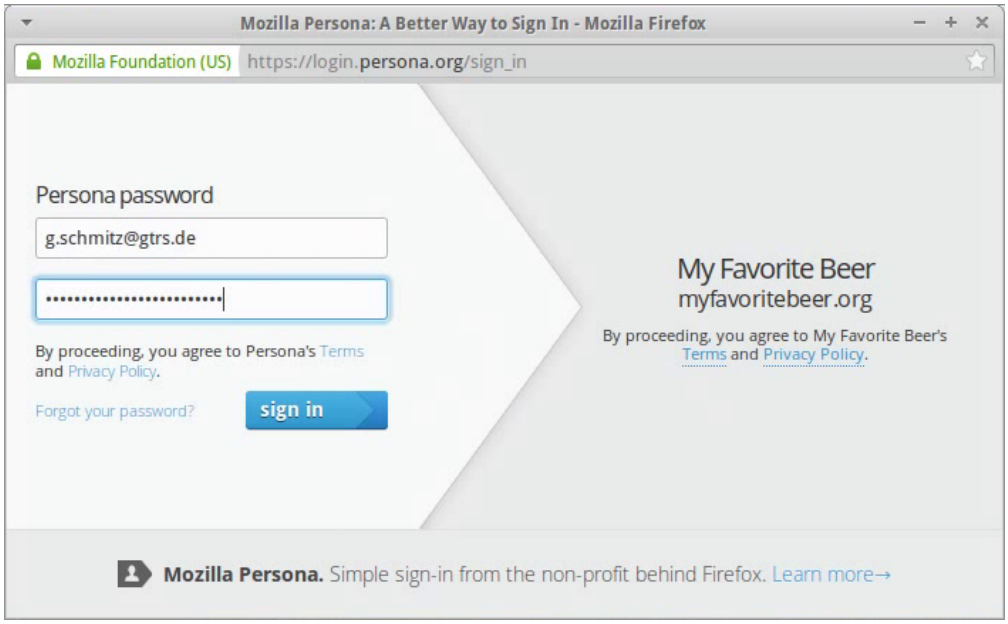


(a) A user visits an RP (a demo RP provided by Mozilla) and is not logged in. The user is offered a button "Sign in with Persona" to log in using BrowserID.



(b) After the user clicked on the login button at the RP, the login dialog opens and the user is prompted to enter her email address.

Figure 4.1.: A user's perspective of a typical BrowserID login flow.



(c) The user's email provider does not support BrowserID. BrowserID is running in secondary mode and uses the fallback IdP to authenticate the user. This interaction is seamlessly integrated into the login dialog.

(d) After the user has entered her credentials into the login dialog, the dialog closes and the user is logged in at RP.

Figure 4.1.: A user's perspective of a typical BrowserID login flow (cont'd).

To enable users of popular email providers, such as Yahoo and Gmail, to use BrowserID natively, Mozilla developed so-called identity bridges. An identity bridge is a Web application that connects BrowserID to other SSO services, such as OpenID 2.0 [FR+07], reducing the effort needed for these providers to implement BrowserID. The identity bridge is (typically) installed at the email provider and implements all IdP-specific BrowserID interfaces. Two identity bridge implementations have been provided by Mozilla, namely Sideshow [Moz15b] (used for Gmail identities) and BigTent [Moz15a] (used for Yahoo identities). For Yahoo and Gmail, however, Mozilla deployed the identity bridges not at the email provider, but in Mozilla's data center instead. The BrowserID implementation explicitly checked whether a user wanted to use a Yahoo or Gmail address and redirected this user to Mozilla's own identity bridge services.

In the remainder of this chapter, we first, in Section 4.1, provide a high-level overview of the BrowserID system. A more detailed description of the BrowserID implementation covering both, the primary and secondary mode, is then given in Sections 4.2 and 4.3 with further details provided in Appendices D and E. We discuss how we model BrowserID in the WIM in Section 4.4 followed by our analysis of authentication and session integrity properties of both modes in Section 4.5. In this section, we also show various severe attacks which violate these properties. We conclude this chapter in Section 4.6 with our analysis of privacy in BrowserID where we show that BrowserID does not hold its promise to protect the user's privacy.

4.1. General Overview

As usual for SSO systems, the BrowserID system knows three distinct parties: the user, who wants to authenticate herself using a browser, the RP to which the user wants to authenticate (log in) with one of her email addresses (say, `user@eyedee.me`), and the IdP. If the IdP (in this example, `eyedee.me`) supports BrowserID directly, it is called a *primary IdP*. Otherwise, a Mozilla-provided service, the so-called *secondary IdP*, takes the role of the IdP as a fallback. The specification of the BrowserID protocol [Adi+13], however, does not forsee such a fallback mechanism — it had been added by Mozilla's implementation.

Here, we first present the general idea of the BrowserID protocol before we discuss how this protocol is realized in the primary and the secondary case using native Web features. Both cases can be considered as separate modes of a protocol that has been derived from the original BrowserID protocol specification.

A (primary) IdP provides information about its BrowserID setup in a so-called *support document*, which it provides at a fixed URL derivable from the email domain, e.g., `https://eyedee.me/.well-known/browserid`.[2] This document contains the public key of the IdP

[2]The path prefix `/.well-known/` is reserved for application-specific interfaces that use fixed paths for each do-

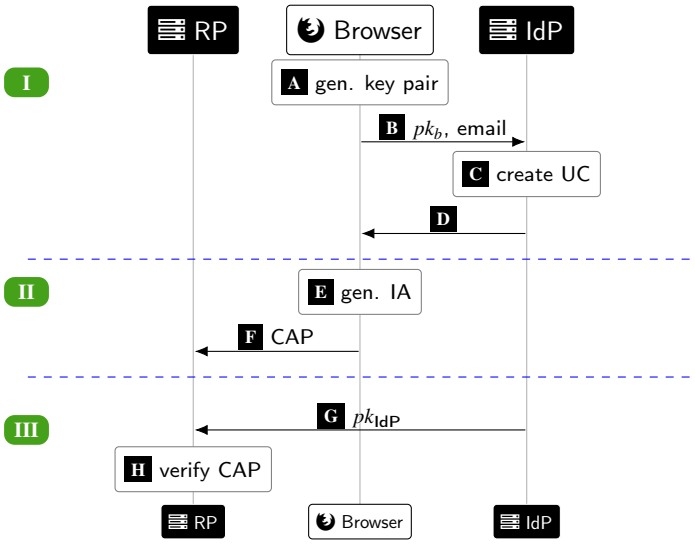


Figure 4.2.: BrowserID high-level overview.

and a set of (URL) paths that are used for interaction with the IdP.

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

The BrowserID authentication process (with a primary IdP) consists of three phases (see Figure 4.2 for an overview): I provisioning of the UC, II CAP creation, and III verification of the CAP.

In Phase I, (the browser of) the user creates a public/private key pair A. She then sends her public key as well as the email address she wants to use to log in at some RP to the respective IdP B. The IdP now creates the UC (given that the user is authenticated at the IdP) C. The UC is then transferred to the user D.

main [RFC5785]. Here, BrowserID uses such a path to discover meta data for a domain.

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

In Phase (III), the RP checks the authenticity of the CAP. For this purpose, the RP can use an external verification service provided by Mozilla or check the CAP itself as follows: First, the RP fetches the support document of the IdP [G], which is contained the public key of the IdP. Afterwards, the RP checks the signatures of the UC and the IA [H]. If this check is successful, the RP considers the user to be logged in with the given email address and send her some token (e.g., a cookie with a session id), which can be considered as an RP service token in the sense of Section 3.1.

The support document can also be used to delegate the governance of identities under a domain, say A, to a different entity. In this case, the support document only contains an entry *authority* that points to a domain, say B, of the other entity. The entity at domain B then acts as the IdP for all identities under the domain A. The issuer is then B in this case.

4.2. Implementation Details of BrowserID's Primary Mode

We now provide a more detailed description of the BrowserID implementation. Since the system is very complex, with many HTTPS requests, XHRs, and postMessages sent between different entities (servers as well as windows and iframes within the browser), we here describe mainly the phases of the login process without explaining every single message exchange done in the implementation. A more detailed step-by-step description can be found in Appendix D.1. Note that BrowserID's specification of IdPs only describes the interface to BrowserID and does not provide further details about (the inner workings of) IdPs. Therefore, in what follows, we consider a typical IdP, namely the example implementation provided by Mozilla [Moz13].

In addition to the parties mentioned so far, the actual BrowserID implementation uses another party, Mozilla's `login.persona.org` (LPO). Among others, LPO (as a trusted third party) provides HTML and JavaScript files that, to ensure their integrity, cannot be delivered by either IdP or RP (which might potentially be malicious). An overview of the implementation is given in Figure 4.3, which we describe below. For brevity of presentation, several messages and components, such as the CIF (see below), are omitted in the figure (see Figure D.1 on Pages 176 and 177 for a more detailed version of Figure 4.3).

Windows and iframes in the Browser. By *RP-Doc* we denote the window containing the document loaded from some RP, at which the user wants to log in with an email address hosted by some IdP. RP-Doc typically includes JavaScript from LPO and contains a button "Login with

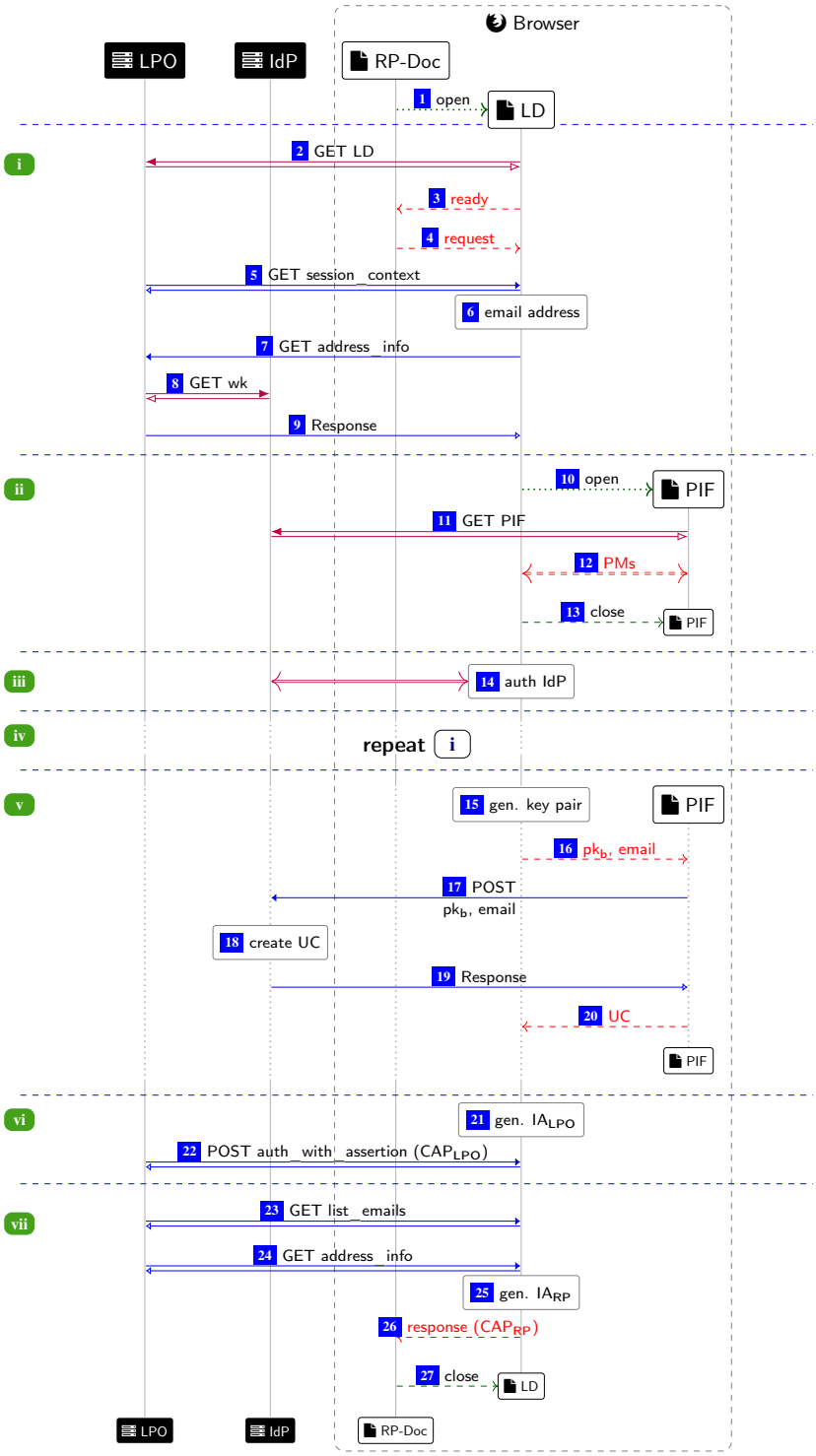


Figure 4.3.: Simplified BrowserID implementation overview (primary mode). The CIF has been omitted for brevity. (See Page 11 for notation.)

BrowserID". The LPO JavaScript running in RP-Doc opens an auxiliary window called the *login dialog* (LD). Its content is provided by LPO and it handles the interaction with the user. During the login process, a temporary (invisible) iframe called the *provisioning iframe* (PIF) can be created in the LD. The PIF is loaded from IdP.[3] It is used by LD to communicate (cross-origin) with the IdP via postMessages. To this end, the PIF uses a standardized API provided by a JavaScript library for communication with the LD (as well as the CIF, see below). Using this API, the PIF acts as the interface between application logic of LPO and application logic of the IdP. (Similar, communication using XHRs with Cross-Origin Resource Sharing (CORS) [Ann14; WHA19a] would also be feasible for cross-origin communication between the LD and the IdP directly, but is not part of the BrowserID protocol.) Temporarily, the LD may navigate itself to a Web page at IdP to allow for direct user interaction with the IdP. We then call this window the *authentication dialog* (AD).[4]

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

Now, the login flow continues with Phase (iv), which basically repeats Phase (i). The user, however, is not prompted for her email address (it has previously been saved in the localStorage under the origin of LPO along with a nonce, where the nonce is stored in the sessionStorage). In Phase (v), which essentially repeats Phase (ii), the PIF detects that the user is now authenticated to the IdP and the provisioning phase is started ((I) in Figure 4.2): The user's keys are created by LD and stored in the localStorage under the origin of LPO. The PIF forwards the certification request to the IdP, which then creates the UC and sends it back to the PIF. The PIF in turn forwards it to the LD, which stores it in the localStorage under the origin of LPO.

[3] Information about the URL of the PIF is contained in the support document of the respective IdP.
[4] Similar to the PIF, information about the URL of the AD is contained in the support document of the IdP.

In Phases (vi) and (vii), mainly the IA is generated by LD for the origin of RP-Doc and sent (together with the UC) to RP-Doc ((II) in Figure 4.2). In the localStorage, LD stores that the user's email address is logged in at RP. Moreover, to log the user in at LPO, LD generates an IA for the origin of LPO and sends the UC and IA to LPO.

LPO Session. LPO establishes a session with the browser by setting a cookie `browserid_state` (in Step [5] in Figure 4.3) on the client-side. LPO considers such a session authenticated after having received a valid CAP (in Step [22] in Figure 4.3). In future runs, the user is presented a list of her email addresses (which is fetched from LPO) in order to choose one address. Then, she is asked if she trusts the computer she is using and is given the option to be logged in for one month or "for this session only" (*ephemeral* session). In order to use any of the email addresses, the user is required to authenticate to the IdP responsible for that address to get an UC issued. If the localStorage (under the origin LPO) already contains a valid UC, then, however, authentication at the IdP is not necessary.

Automatic CAP Creation. In addition to the interactive login presented above, BrowserID also contains an automatic, non-interactive way for RPs to obtain a freshly generated CAP: During initialization within RP-Doc, an invisible iframe called the *communication iframe* (CIF) is created inside RP-Doc. The CIF's JavaScript is loaded from LPO and behaves similar to LD, but without user interaction. The CIF automatically issues a fresh CAP and sends it to RP-Doc under specific conditions: among others, the email address must be marked as logged in at RP in the localStorage. If necessary, a new key pair is created and a corresponding new UC is requested at the IdP. For this purpose, a PIF is created inside the CIF (see Figure D.1 on Pages 176 and 177 for a detailed version of Figure 4.3 including the usage of the CIF).

Logout. We have to differentiate between three ways of logging out: an RP logout, an LPO logout, and an IdP logout.

An RP logout is handled by the CIF after it has received a *logout* postMessage from RP-Doc. The CIF then changes the localStorage such that no email address is recorded to be logged in at RP.

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

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

Sideshow and BigTent. As already mentioned above, Mozilla implemented so-called identity bridges, namely Sideshow and BigTent to connect Gmail and Yahoo users to the BrowserID ecosystem using other SSO protocols. While Sideshow has been created as a Gmail specific iden-

tity bridge, BigTent was created to support multiple providers, including Yahoo and Microsoft Hotmail, for example. In practice, BigTent only supported Yahoo. Both, Sideshow and BigTent initially used the SSO protocol OpenID 2.0 to interact with Gmail and Yahoo. When Google shut down its OpenID service [Goo16], Sideshow was changed to use OAuth 2.0 [RFC6749] and later OpenID Connect 1.0 [Sak+14] to authenticate users. In our description below, we focus on the OpenID 2.0 variant of Sideshow; BigTent works in a very similar fashion. Technical details on the communication between OpenID and Sideshow/BigTent can be found in Appendix D.2.

To "bridge" between BrowserID and the OpenID service, Sideshow acts as a "translator" and is placed in between BrowserID and the email provider (Gmail) running OpenID. That is, BrowserID uses Sideshow as an IdP. Sideshow translate requests from BrowserID to requests to the email provider's OpenID interface.

All BrowserID protocol steps that would normally be carried out by the IdP are now handled by Sideshow (i.e., the Sideshow server). For this purpose, Sideshow serves the provisioning URL (for the PIF) and the authentication URL used in (iii). It maintains a session with the user's browser. This session is considered to be authenticated if the user has successfully authenticated to Sideshow using OpenID. In this case, Sideshow's PIF document may send public keys to Sideshow. Sideshow then creates a UC for the identity it believes to be logged in. If the session at Sideshow is not authenticated, the user will first be redirected to the Sideshow authentication URL. Sideshow's authentication document will redirect the user further to the OpenID URL at Gmail. This URL contains an authentication request encoding that Sideshow requests an *OpenID assertion* that contains an email address. In general, such an assertion is a list of attribute name/value pairs (partially) MACed by Gmail with a temporary symmetric key known only to Gmail; an additional attribute, `openid.signed`, in such an assertion encodes which attribute name/value pairs have actually been MACed and in which order. The user now authenticates to Gmail. Then, Gmail issues the requested OpenID assertion and redirects the browser to Sideshow with the assertion in the URL parameters. Sideshow then sends the OpenID assertion to Gmail in order to check its validity. If the OpenID assertion is valid, i.e. the MAC over the attributes listed in `openid.signed` verifies, Sideshow considers its session with the user's browser to be authenticated for the email address contained in the OpenID assertion.

4.3. Implementation Details of BrowserID's Secondary Mode

If an email provider does not directly support BrowserID (indicated by the absence of the support document), LPO serves as the so-called *secondary IdP* (sIdP), i.e., LPO takes the role as the IdP

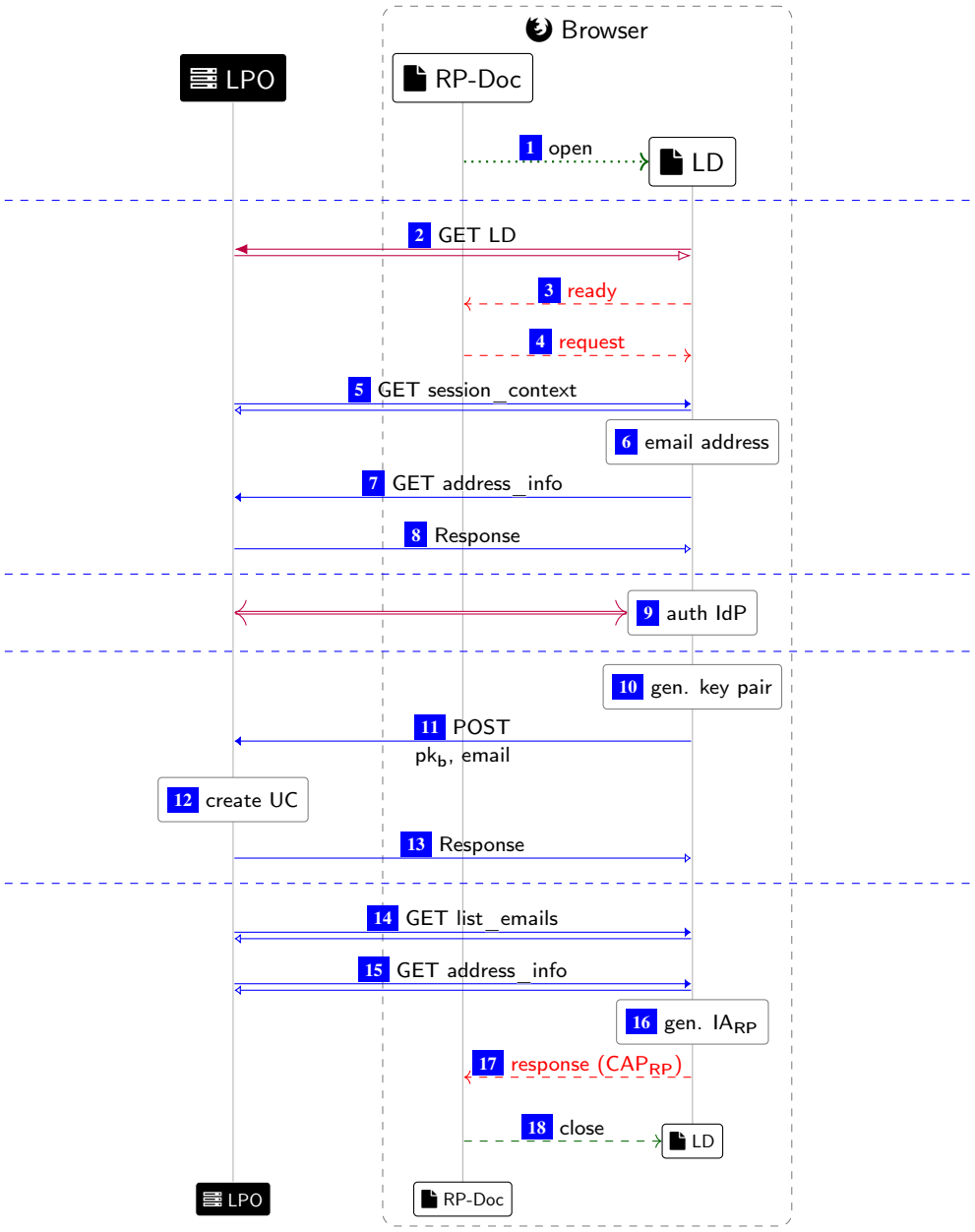


Figure 4.4.: Simplified BrowserID implementation overview (secondary mode). The CIF has been omitted for brevity. (See Page 11 for notation.)

for this email provider. As LPO does not have access to the email provider's user database, the user has to register at LPO. That is, she creates an account at LPO where she can register one or more email addresses to be used as identities. She has to prove ownership of all email addresses she registers. (LPO sends URLs to each email address, which then have to be opened by the user.)

When secondary mode is used, the phases ii – vi (see Figure 4.3) are not needed as LPO replaces the IdP and the actions previously performed by IdP and LPO are now carried out by LPO alone. The user is prompted to enter her password directly into LD. If the password is correct, LPO now considers the session with the browser to be authenticated. LPO will then issue UCs on behalf of the email provider. We note that, for automatic CAP creation, the CIF (see Section 4.2) is still used. The PIF and the AD, however, do not exist. Moreover, in the primary mode, the behavior of the CIF and the LD is more complex than in the secondary mode. For example, in the primary mode, just like the LD, the CIF might contain a PIF (as an iframe within the CIF iframe) and interact with it via postMessages. Altogether, the secondary IdP case requires much less communication between parties/components and trust assumptions in a setting with secondary mode are simpler as there is only one IdP, namely LPO. As LPO is always considered to be honest, there are no malicious IdPs.

In Figure 4.4, we provide an overview of a typical BrowserID login flow in secondary mode. A detailed step-by-step description can be found in Appendix E.1.

4.4. Modeling BrowserID in the WIM

We model both modes of BrowserID, primary and secondary, as separate SSO Web systems. This approach, as explained above, is motivated by the fact that both modes constitute two different protocols. Both modes, however, share the high-level idea of BrowserID and hence, share some parts in the implementation. LPO evaluates at the beginning of each protocol run, whether the primary or the secondary mode is used for a specific user id: LPO tries to retrieve the support document from the domain of the user's email address. If the support document is available, LPO continues in primary mode, else LPO falls back to the secondary mode and serves as the IdP for this identity. The secondary mode, however, is not part of the original BrowserID specification [Adi+13], but was added by Mozilla as a temporary solution for marketing and bootstrap purposes to allow virtually all users to be instantly able to benefit from BrowserID [Mil11]. We further discuss this division in Section 4.7 below.

4.4.1. Primary Mode

We model the BrowserID system with primary IdPs as an SSO Web system (as defined in Section 3.1). Note that while in Section 4.2 we give only a brief overview of this mode, our modeling and analysis considers the complete system with primary IdPs, where we have extracted the model from the BrowserID source code [Moz13].

We call an SSO Web system $\mathcal{BID}^p = (\mathcal{W}^p, \mathcal{S}, \mathsf{script}, E_0)$ a *BrowserID primary mode Web system* if it is of the form as described in Appendix D.3. In what follows, we outline the components of such a Web system.

Following Definition 4, we define the system $\mathcal{W}^p$ to be composed of the (pairwise disjoint) subsets Web, Net, B, RP, IDP, DNS, and Other. The set $\mathsf{Other} = \{\mathsf{LPO}\}$ consists of one Web server for LPO. For the analysis of authentication and session integrity, we use a *BrowserID primary mode Web system with a network attacker* that is an BrowserID primary mode Web system with $\mathsf{Web} = \emptyset$ and $\mathsf{Net} = \{\mathsf{attacker}\}$. For the analysis of privacy, we consider a *BrowserID primary mode Web system for privacy analysis* that follows Definition 16.[5]

The set $\mathcal{N}$ of nonces is partitioned into four (disjoint) sets: The (infinite) set $N^{\mathcal{W}^p}$ contains the nonces that are available for each DY process in $\mathcal{W}^p$ ($N^{\mathcal{W}^p}$ is set as a sequence in the initial configuration of the system), the set K_{TLS} contains the keys that will be used for TLS encryption, the set K_{sign} contains the keys that will be used by IdPs for signing UCs (signing keys),[6] and the set $\mathsf{Secrets} \subseteq \mathcal{N}$ is the set of passwords (secrets) the browsers share with the identity providers. The nonces in K_{TLS}, K_{sign}, and Secrets are distributed according to their purpose over the initial states of the processes (see below).

The set IPs contains one (IP) address each for LPO, every relying party in RP, every identity provider in IDP, every DNS server in DNS, every browser in B, every Web attacker in Web, and every network attacker in Net.[7] By addr we denote the corresponding mapping from a process to its address. The set Doms contains one domain for LPO, one for every relying party in RP, a finite set of domains for every identity provider in IDP, and a finite set of domains for attackers. Browsers (in B) and DNS servers (in DNS) do not have a domain. By dom we denote the corresponding mapping from a process to its domains. Each domain is assigned a unique private key from K_{TLS} and for each domain, the "owner" (according to dom^{-1}) is given the respective private key in its initial state. For each of these keys, the public key (i.e., $\mathsf{pub}(k)$ for $k \in K_{\mathrm{TLS}}$) is given to (the initial state of) browsers, RPs, and attackers in the form of a dictionary indexed by the key's domain.

[5] As we will see in Section 4.6 below, BrowserID is not IdP-private. Hence, we cannot show this property for BrowserID and omit a full formal specification of the BrowserID primary mode Web system for privacy analysis.
[6] We call the public counterpart of a (private) signing key ($k \in K_{\mathrm{sign}}$) a verification key ($\mathsf{pub}(k)$).
[7] A network attacker can, by definition, also use any other IP address.

(User) identities are used as specified in Definition 1, which also matches the nature of email addresses. Each browser $b \in \mathsf{B}$ is assigned a (finite) set of identities and their respective secrets (out of Secrets). Identities and secrets are unique to a browser, i.e., browsers have disjoint sets of identities and disjoint sets of secrets.

As explained in Section 3.1, we define ownerOfSecret to be a function from secrets (passwords) to browsers and secretOfID to be a function from identities to secrets. ownerOfID is defined as a "shortcut" that yields the browser that owns an identity. An identity is governed by the owner of the domain contained in the identity, i.e., we define the governor to be a function that resolves the domain of the identity to the process to which the domain was assigned to (see also above). A user can have multiple identities that are governed by the same IdP. In the initial state of each IdP, we provide a list of pairs of identities and corresponding secrets that are governed by the IdP. Similar, we set the initial state of browsers such that browsers provide secrets to scripts that run under a secure origin of their identities' governor.[8]

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

The set E_0 only contains the trigger events for each address as sketched in Section 2.3 and specified in Definition 61 in Appendix A.8.

As explained above, BrowserID relies on user certificates, identity assertions, and certificate assertion pairs. We formally define these as follows:

Definition 18. A (valid) *user certificate* (UC) uc for a user u with email address $id = \langle name, d \rangle$ and public key (verification key) $\mathsf{pub}(k_u)$, where $d \in \mathsf{dom}(y)$ is a domain of the governor y of id and k_u is the private key (signing key) of u, is a message of the form

$$uc = \mathsf{sig}(\langle \langle name, d \rangle, \mathsf{pub}(k_u) \rangle, \mathsf{signkey}(y)) \ .$$

A (valid) *identity assertion* (IA) ia for an origin o (e.g., $\langle \texttt{example.com}, \texttt{S} \rangle$) signed with the key k_u is a message of the form $ia = \mathsf{sig}(o, k_u)$.

A *certificate assertion pair* (CAP) is of the form $\langle uc, ia \rangle$, with uc and ia as above.[9] ◇

The (sigining) keys for UCs (out of K_{sign}) are distributed in the initial states of IdPs, RPs, and LPO as follows: Each IdP is given one private signing key that IdPs uses to sign all UCs for

[8] Recall that browsers provide a list of secrets to a script based on the origin of this script's document.

[9] Note that the time stamps are omitted both from the UC and the IA. This models that both certificates are valid indefinitely. In reality, they are valid for a certain period of time, as indicated by the time stamps. So our modeling is a safe overapproximation.

identities it governs. Every RP and LPO are given a dictionary that maps each domain that is assigned to an IdP to the respective public key ($\mathsf{pub}(k)$ for $k \in K_{\text{sign}}$) of that IdP.

We now describe the processes in $\mathcal{W}$ as well as the scripts in $\mathcal{S}$. In Appendix D.3, we provide a detailed formal specification of these processes and scripts.

Browsers. Each $b \in \mathsf{B}$ is a Web browser as described in Section 2.5. As explained above, the initial state contains all secrets owned by b and a list of all identities owned by this browser. Note that to the browser we do not provide information about which secret is assigned to which identity (except for the assignment to the correct governor's origin), allowing a user to confuse her passwords at an IdP. Still, each secret is "valid" for only one identity at this IdP.

See Appendix D.3.5 for further details on browsers in $\mathcal{W}^p$.

LPO. LPO (formally, $\mathsf{LPO} \in \mathsf{Other}$) is a Web server that serves important scripts (*script_lpo_cif* and *script_lpo_ld*) and manages user sessions (by assigning cookies to browsers). As we consider the primary mode, these user sessions are mainly used for XSRF protection at LPO, i.e., LPO assigns a nonce, the XSRF token, to each session. Each login attempt sent to LPO needs to include this token (together with the respective cookie) in order to be accepted by LPO. Note that LPO also acts as an RP in primary mode. In practice, the "RP service" provided by LPO is to present the user a list of her identities, which she has used before. In our model, we do not use this (potentially incomplete) list and allow for logins using any of the user's identities (based on the list of identities contained in the browser). The selection of an identity, however, is handled by the script *script_lpo_ld* (see also below). Still, we include the authentication at LPO in our model as ephemeral sessions (i.e., the respective cookie is set to be removed by the browser when the browser is closed, i.e., closecorrupted in our model).

In its initial state, LPO is given the (private) TLS key for its domain, a dictionary of (public) verification keys, i.e., for each domain owned by an IdP, the respective public key to verify UC signatures.

Note that LPO cannot become corrupted. If LPO would be corruptible, all security and privacy properties would be trivially broken.

Client sessions at LPO. Any party can establish a *(Web) session* at LPO (based on a session identifier stored in a cookie). Such a session can either be authenticated or unauthenticated. Roughly speaking, a session becomes authenticated if a client has provided a valid CAP (for the origin of LPO) to LPO during the session. LPO manages groups of identities, i.e., lists of email addresses. If a user authenticates a session using any identity in the group, she is authenticated for all identities in the group. An authenticated session can (non-deterministically) *expire*, i.e. the authenticated session can get unauthenticated or it is removed completely. Expiration of sessions is modeled as a non-deterministic choice when a trigger event is processed. Such an

expiration is used to model a user logout or a session expiration caused by a timeout.

Each session is identified by a nonce (issued by LPO) and is associated with some XSRF token, which, as mentioned above, is also a nonce issued by LPO. LPO stores all information about established sessions in its state as a dictionary indexed by the session identifier. In this dictionary, for every session, LPO stores a pair containing the XSRF token and, in authenticated sessions, the sequence of all identities associated with the secret provided for this (Web) session, or, in unauthenticated sessions, the empty sequence $\langle\rangle$ of identities. On the client side (typically a browser) LPO places, by the usage of appropriate headers in its HTTPS responses, a cookie named `browserid_state` carrying the session identifier (a nonce). This cookie is flagged to be a session, httpOnly, and secure cookie.

HTTPSRequests to LPO. LPO answers only to certain requests (listed below). In reality, all such requests have to be received over HTTPS, and all responses sent by LPO contain the `Strict-Transport-Security` header. We safely overapproximate in omitting these two requirements in the model.

`GET /cif`. LPO replies to such a request by providing the script *script_lpo_cif* that models the CIF.

`GET /ld`. LPO replies to such a request by providing the script *script_lpo_ld* that models the login dialog (LD).

`GET /ctx`. Requests of this form query the current user's session context information from LPO. The response body is of the form $\langle loggedIn, xsrfToken\rangle$, where *loggedIn* is $\top$ or $\bot$, depending on whether the user is logged in at LPO or not, and *xsrfToken* is the token that the client is supposed to include into the auth request (see below).

`POST /auth`. With such a request, a client can log into LPO. The client has to provide a a CAP and an XSRF token. If the CAP is valid and issued for the origin of LPO and the XSRF token matches the one recorded in the session at LPO, LPO considers the user to be authenticated and modifies the session record in its state accordingly.

See Appendix D.3.6 for full details about LPO.

IdPs. Each IdP $i \in \mathsf{IdP}$ is a Web server. IdPs are modeled following the example implementation provided by Mozilla and follow the security considerations in [Moz18] (XSRF protection, e.g., by checking origin headers, and using HTTPS only with STS enabled). As outlined in Section 4.2, users can authenticate to the IdP with their credentials. IdP tracks the state of the users with sessions. Authenticated users can receive signed UCs from the IdP.

In their initial state, each IdP is given its (private) TLS keys, the private signing key that is used to sign UCs, and information about the identities this IdP governs. This includes the respective secret for each identity governed by this IdP. To define the initial state, we define 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 set of identities. It is encoded as a mapping of secrets to lists of identities for which these secrets are valid. For example, if the secret $secret_1$ is valid for the identites id_1 and id_2 and the secret $secret_2$ is valid for the identities id_3 and id_4, the $userset^i$ looks as follows:

$$userset^i = [secret_1{:}\langle id_1, id_2\rangle, secret_2{:}\langle id_3, id_4\rangle]$$

Initially, IdPs are honest. When receiving a corrupt message IdPs become corrupted. Similar to the definition of corruption for the browser, IdPs then start sending out all messages that are derivable from their state. This means, that they effectively become Web attackers.

Client sessions at IdPs. When a user authenticates to an IdP, the IdP creates a (Web) session. This session is identified by a session identifier (a nonce) and is recorded (along with the id of the authenticated user) in the IdPs state. As usual, the session identifier is set in the browser as a cookie. This session is only used to authenticate requests for UCs (see below).

HTTPSRequests to IdPs. Similar to LPO above, IdPs answer only to certain requests (listed below). All of these requests have to be received over HTTPS and IdPs set the `Strict-Transport-Security` header in all responses.

`GET *`. An IdP replies to all GET requests (except the one listed below) by providing the script *script_idp_ad* that models the authentication dialog.

`GET /pif`. An IdP replies to such a request by providing the script *script_idp_pif* that models the PIF.

`POST *`. An IdP assumes that all POST requests (except the one listed below) to be authentication requests. If the request contains an `Origin` header for a secure origin of the IdP (for XSRF protection) and the body of the request contains a valid pair of username and password, the IdP creates a client session (see above) and sets the session identifier in a cookie in its response.

`POST /certreq`. An IdP expects such a request to be a signing request of a user's public key. The request must contain a cookie with a valid client session identifier, and, in its body, the user id and the user's public key.[10] Only if the user id in the request matches the id

[10] The public key can be an arbitrary term.

recorded in the client session, the IdP creates a UC for this id and the public key contained in the request. The UC is then sent out in the HTTPS response.

See Appendix D.3.7 for details of IdPs.

RPs. A relying party $r \in \mathsf{RP}$ is a Web server. The definition of R^r follows the description in Section 4.2 and, as IdPs, follow the security considerations in [Moz18]. An RP replies to any `GET` request with the script *script_rp_index* (see below). When receiving an HTTPS `POST` message, RP checks (among others) if the message contains a valid CAP. For this purpose, all signing keys of the identity providers (see below) are contained in the initial state of all RPs. If successful, RP responds with an RP service token n for the identity id, where $id \in \mathsf{ID}$ is the identity for which the CAP was issued and n is a freshly chosen nonce. RP also stores the respective service session in its state.

In a typical flow with one client, r will first receive an HTTP GET request. In this case, it returns the script `script_rp_index` (see Appendix D.3.9 below) and sets the `Strict-Transport-Security` header.

In their initial state, each RP is given its (private) TLS keys and, as explained above for LPO, a dictionary of (public) verification keys. Just like IdPs, RPs are initially honest and can become corrupted.

Service sessions at RPs. Each RP maintains (in its state) a list of service sessions (as described in Section 3.1).

When an RP receives an HTTPS POST request with a CAP, r checks that the UC and IA are valid and matching. If the check is successful, r creates a new service session for the identity i, $\langle n, i \rangle$, and sends the service token to the browser. The RP keeps a list of such tokens in its state.

HTTPSRequests to RPs. As LPO and IdPs, RPs answer only to certain requests (listed below). Again, all of these requests have to be received over HTTPS. and RPs also set the `Strict-Transport-Security` header in all responses.

`GET *`. An RP delivers the script *script_rp_index* in response to all GET requests. This script models the Web page of RP.

`POST *`. If an RP receives a POST request, the RP verifies the `Origin` header for XSRF protection. This header must be set to a secure origin of the RP. The RP expects that the request contains a CAP, checks the signatures, and (if the signatures are correct) finally logs the user in under the identity stated in the CAP by creating a service session and sending the corresponding service token in the response.

See Appendix D.3.8 for details on RPs.

BrowserID scripts. The set $\mathcal{S}$ of the Web system $\mathcal{BID}^p{}_{\text{primary}} = (\mathcal{W}^p, \mathcal{S}, \mathsf{script}, E_0)$ consists of the scripts R^{att}, *script_rp_index*, *script_lpo_cif*, *script_lpo_ld*, *script_idp_pif*, and *script_idp_ad* (a mapping to string representations is defined by script).

The script R^{att} is the attacker script (see Section 2.5) and the formal modeling of the remaining (honest) scripts follows the description below (see Appendix D.3.9 for more details).

All of these scripts are modelled as state machines (as their real-world counterparts) that use the subterm q of their scriptstate to track their state. Hence, the behavior mainly depends on the state q the respective script is in.

Relying party script (script_rp_index). The script *script_rp_index* defines the script of the RP index page. This script aggregates the behaviour of all scripts that — in reality — are included on an RP's index page. In particular, this includes a script that is provided by LPO.[11] In particular, this script is responsible for creating the CIF and the LD iframes/subwindows, whose contents are loaded from LPO. and later use the service token after a successful login.

When run, the script behaves as follows:

$q \equiv$ `init` This is the initial state. The script creates the CIF iframe and then transitions to `receiveCIFReady`.

$q \equiv$ `receiveCIFReady` In this state, the script expects a `cifready` postMessage from the CIF iframe with the sender origin of LPO. The script chooses some identity, $\langle\rangle$, or $\bot$ and sends this in a `loaded` postMessage to the CIF iframe with receiver's origin set to the origin of LPO.[12] It then transitions to the state `default`.

$q \equiv$ `default` In this state, the script chooses non-deterministically between (1) opening the LD subwindow and then transitioning to the same state or (2) handling one of the following postMessages (identified by their first element):

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

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

[11] Recall that LPO is always honest in our setting (otherwise all security properties would be trivially broken) and that the script provided by LPO is always loaded from a secure origin of LPO.

[12] From the point of view of the real scripts running at RP either some identity is considered to be logged in (e.g. from some former "session"), or no one is considered to be logged in ($\langle\rangle$), or the script *script_rp_index* does not know if it should consider anyone to be logged in ($\bot$). This is overapproximated here by allowing *script_rp_index* to choose non-deterministically between these cases.

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

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

$q \equiv$ `dlgClosed` In this state, the script sends a `loggedInUser` postMessage to the CIF and transitions to the `loggedInUser` state.

$q \equiv$ `loggedInUser` In this state, the script sends a `dlgCmplt` postMessage to the CIF and transitions to the `sendCAP` state.

$q \equiv$ `sendCAP` In this state, the script sends the CAP to RP as a `POST` XHR and then transitions to the `receiveServiceToken` state.

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

LPO login dialog script (script_lpo_ld). The script *script_lpo_ld* provides the functionality of the BrowserID login dialog. The LD controls the BrowserID login flow and also takes care of requesting and using UCs.

This script (as well as the script *script_lpo_cif*) makes use of Web storage (localStorage and sessionStorage) for tracking a login flow of a user across several instances of (documents containing) the script. The Web storage (under the origin of LPO) is organized as follows:

The localStorage is a dictionary. There are two types of entries in this dictionary: Under the key `siteInfo`, a dictionary is stored which has origins as keys and identities as values. An entry in this dictionary indicates that the user is logged in at the referenced origin with a certain identity. The second type of entry has a nonce as a key. The value is an email address (identity). This models the email address a user entered in the LD before being navigated away to the AD. The nonce is also stored in the sessionStorage (see below).

For example, if the user is logged in at $\texttt{domain}_{\texttt{RP1}}$ and $\texttt{domain}_{\texttt{RP2}}$ with id_I and at $\texttt{domain}_{\texttt{RP3}}$

with id_2 (using HTTPS), the localStorage can be as follows:

$$\begin{array}{lll} \langle & \langle \texttt{siteInfo}, \langle & \langle\langle \texttt{domain}_{\texttt{RP1}}, \texttt{S}\rangle, id_1\rangle, \\ & & \langle\langle \texttt{domain}_{\texttt{RP2}}, \texttt{S}\rangle, id_1\rangle, \\ & & \langle\langle \texttt{domain}_{\texttt{RP3}}, \texttt{S}\rangle, id_2\rangle\rangle\rangle, \\ & \langle n_1, id_1\rangle, & \\ & \langle n_2, id_3\rangle\rangle & \end{array}$$

The nonces n_1 and n_2 shown above each refer to a login flow in which the user entered an email address in the LD. The sessionStorage is supposed to refer to one of these entries as the entry for the "current" login flow. Mozilla implemented the usage of Web storage this way to support multiple concurrent login flows.[13] The sessionStorage is also a dictionary. Here, it may only contain one key, `idpnonce`. Its value is a nonce (e.g., n_1 or n_2 in the example above) which references the latest email address entry in the localStorage.

We now describe the states of *script_lpo_ld*.

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

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

$q \equiv$ `receiveContext` In this state, the script expects an XHR response as input containing the session context. This context is saved as the current context in the scriptstate. The script checks if an `idpNonce` is recorded in the sessionStorage. The presence of this nonce indicates that there was a run of `script_lpo_ld` in the same window previously. Indexed by this nonce, there can be an email address (identity) recorded in the localStorage which is then copied to the scriptstate. Otherwise an email address is non-deterministically choosen (and copied to the scriptstate) out of the email addresses owned by the browser.

The script now always issues the command to create an iframe, the PIF. The URL for the PIF is determined by the domain of the email address now recorded in the state. The script then transitions to the state `startPIF`.

$q =$ `startPIF` In this state, the script waits for a postMessage from the PIF containing a `ping` message. If such a message is received and the sender's window and origin match the PIF,

[13] Recall that sessionStorage is (in addition to origin) also bound to the top-level window of the current document.

the script sends a `pong` message back to the PIF and transitions to the state `runPIF`.

$q =$ `runPIF` This is the state in which `script_lpo_ld` interacts with the PIF. This state handles all postMessages the LD expects to receive from the latest PIF (as recorded in `PIFindex` in its state). If the postMessage received was sent from the PIF's window and the PIF's origin, it behaves as follows, depending on the first element of the received postMessage:

postMessage `beginProvisioning` The script responds with a postMessage to the PIF containing the email address of the identity which is to authenticate to the relying party (as recorded in the LD's state).

postMessage `genKeyPair` The script creates a fresh key pair (i.e. the LD chooses a fresh nonce) and sends the public key contained in an postMessage to the PIF.

postMessage `registerCertificate` The script stores the UC received in this postMessage in the LD's state. If the context contained in the scriptstate indicates that the browser is authenticated to LPO, the script transitions to the state `createCAPforRP`. Otherwise, the script transitions to the state `createCAPforLPO`.

postMessage `raiseProvisioningFailure` This message indicates that the user is not logged in at the IdP. The script now chooses a fresh nonce, the so-called *idpNonce*, which is stored in the sessionStorage. In the localStorage, this nonce is used as a key under which the email address is stored, the LD is currently trying to get an UC for. The script navigates the window it is running to the authentication path at the identity provider responsible for the email address.

$q =$ `createCAPforLD` In this state, the script creates an IA for LPO, combines it with the UC (stored in the scriptstate) to a CAP and sends the CAP to LPO in an XHR. The nonce identifying the XHR is stored as `refXHRLPOauth` in the scriptstate.

$q =$ `receiveLPOauthresponse` In this state, the script expects the response to the XHR identified by the nonce `refXHRLPOauth`. If the response indicates a successful authentication at LPO, the context recorded in the scriptstate is changed accordingly and the script transitions to the state `createCAPforRP`.

$q =$ `createCAPforRP` In this state, the script creates an IA for the request origin (as recorded in the scriptstate), combines the IA with the UC to a CAP, and sends the CAP in a postMessage to its parent restricting the receiver to the request origin. The script records in the localStorage that the email address it is currently using is logged in at the request origin. The script then transitions to the state `null`.

$q \equiv$ `null` In this state, the script does nothing.

LPO communication iframe script (script_lpo_cif). The script *script_lpo_cif* provides the functionality of the CIF, which takes a similar role as the LD but without user interaction. We provide a detailed description of this script in Appendix D.3.9.

IdP authentication dialog script (script_idp_ad). The script *script_idp_ad* provides the functionality of the authentication dialog provided by IdPs. Recall that this script is supposed to run in the same window as the LD after *script_lpo_ld* has navigated the LD window to the AD.

The script non-deterministically chooses between sending authentication data to the IdP (i.e. its origin) via an XHR or navigating the window to an URL at LPO which servers *script_lpo_ld*. This script does not make use of any scriptstate.

IdP provisioning iframe script (script_idp_pif). This script acts as a proxy between the LD (or CIF) and the IdP server. The states of this script are as follows:

$q =$ `init` This is the initial state. Its only transition handles no input and outputs a postMessage `ping` to its parent window, which has to have the origin of LPO, and transitions to `waiting`.

$q =$ `waiting` In this state, the script expects a postMessage containing either `ping` or `pong`, which has to be sent by the parent window from the origin of LPO. If such a postMessage has been received, the script transitions to `default`.

$q =$ `default` In this state, the script chooses an action non-deterministically out of the following:

`beginprovisioning` The script sends a postMessage to the parent window, which has to have the origin of LPO, indicating that the provisioning process of a UC should start. A fresh nonce is chosen, stored in the scriptstate, and included in this postMessage. The postMessage requests the email address of the user from the receiver. The address is to be sent to the PIF in a postMessage which is identified by the nonce in the request.

`genkeypair` The script sends a postMessage to the parent window, which has to have the origin of LPO, indicating that a new key pair should be generated. This postMessage requests the public key of this fresh key pair. As above, a nonce is included to identify the response corresponding to the request.

`registercert` The script sends a postMessage containing a UC to the parent window, which has to have the origin of LPO. This postMessage is only sent if the script has received a UC before.

`raisefailure` The script sends a postMessage to the parent window, which has to have the origin of LPO, indicating that the browser is currently not authenticated to the identity provider.

`requestuc` The script sends an XHR to the origin of the current document if the scriptstate contains at least one email address and one public key. The message contains a non-deterministically chosen email address and a public key (from the scriptstate). The nonce identifying this XHR is non-deterministically chosen and stored in the scriptstate.

`handleresponse` The script chooses non-deterministically one of its script inputs and distinguishes if this input is a postMessage or an XHR response.

If the chosen input is a postMessage, it is checked if the postMessage was sent by the parent window and if this window has the origin of LPO. If this check is successful, it is checked if the message contains a nonce, which was previously been recorded in the scriptstate. If this nonce indicates that this message is a response to a `beginProvisioning` postMessage, the second part is assumed to contain an email address. This address is then recorded in the scriptstate. If the nonce indicates that this message is a response to a `genKeyPair` postMessage, the second part is assumed to contain a public key. This public key is then recorded in the scriptstate.

If the chosen input is an XHR response, it is checked if the nonce identifying the XHR is recorded in the scriptstate. If this is the case, the message is assumed to contain an UC. The content of the message is stored in the scriptstate.

4.4.2. Secondary Mode

Our model for BrowserID's secondary mode is constructed in a similar way as the model for the primary mode above. For presentation, we here only give a brief overview of the model for this mode. Full details are contained in Appendix E.2.

As discussed in Section 4.3 above, this mode uses some shortcuts and simplifications of the BrowserID protocol and effectively constitutes a different protocol. In the secondary mode, LPO takes the role of all IdPs and the BrowserID login dialog (provided by LPO) also takes care of authenticating the user and provisioning UCs. Hence, there is neither a provisioning iframe nor a separate authentication dialog.

Again, we model BrowserID in secondary mode as an SSO Web system. We call such a Web system $\mathcal{BID}^s = (\mathcal{W}^s, \mathcal{S}, \mathsf{script}, E_0)$ a *BrowserID secondary mode Web system*.The system $\mathcal{W}^s$ is consists of the subsets Web, Net, B, RP, IDP, DNS, Other. The set $\mathsf{IDP} = \{\mathsf{LPO}\}$ consists of

one Web server for LPO, while the set Other is empty. As in the primary mode, for authentication and session integrity, we consider a network attacker. Privacy does not apply to the secondary mode.

The set $\mathcal{N}$ of nonces is defined similar to the primary mode. As all identities are (in contrast to the primary mode) governed by LPO, we only have one IdP. Hence, the set K_{sign} only contains one nonce, k^{LPO}, that is used by LPO to sign UCs. As UCs are only issued by LPO (on behalf of the owners of the respective domains), a UC is of the form $uc = \mathsf{sig}(\langle i, \mathsf{pub}(k_u)\rangle, k^{\mathsf{LPO}})$ with k_u being the private key (signing key) of some user with the identity u.

As in primary mode, where a user has a different secret (password) for each identity even at the same IdP, in our model for the secondary mode, a user also has a different secret for each identity (i.e., the user needs to log in at LPO for each identity separately by providing the correct secret).[14]

The set $\mathcal{S}$ contains four scripts (with their string representations defined by script): the honest scripts *script_RP_index*, *script_LPO_cif*, and *script_LPO_ld* as well as the malicious script R^{att}.

4.5. Authentication and Session Integrity of BrowserID

In this section, we present the analyses of both modes of the BrowserID system with respect to the authentication and session integrity properties.

As mentioned in the introduction, during the analysis of BrowserID it turned out that these properties are not satisfied and that in fact there are attacks on BrowserID. We confirmed that all attacks work on the actual implementation of BrowserID. We reported all attacks to the developers at Mozilla, who acknowledged them. We have been awarded several bug bounties by Mozilla. In Section 4.5.1, all attacks are presented along with fixes. (Our BrowserID models presented in Appendices D.3 and E.2 contain these fixes.) In Section 4.5.2, we prove that the fixed BrowserID systems satisfies both authentication properties.

4.5.1. Attacks and Fixes

Here, we present the attacks on BrowserID that we found and propose fixes for all of these problems.

Identity Injection Attack (Primary Mode Only). While trying to prove the security properties of BrowserID with primary IdPs in our model, we discovered a serious attack, which is sketched below and does not apply to the case with secondary IdPs. We confirmed the attack on the actual implementation and reported it to Mozilla [Bug14].

[14] We assume that users are already registered at LPO.

During the provisioning phase (v) (see Figure 4.3), the IdP issues a UC for the user's identity and public key provided in 16. This UC is sent to the LD by the PIF in 20.

If the IdP is malicious, it can issue a UC with different data. In particular, it could replace the email address by a different one, but keep the original public key. This (malicious) UC is then later included in the CAP by LD. The CAP will still be valid, because the public key is unchanged. Now, as the RP determines the user's identity by the UC contained in the CAP, RP issues a service token for the spoofed email address. As a result, the honest user will use RP's service (and typically will be logged in to RP) under an identity that belongs to the attacker, which, for example, could allow the attacker to track actions of the honest user or obtain user secrets.This clearly violates security w.r.t. session integrity.

Proposed fix. Upon receipt of the UC in 20 of Figure 4.3, LD should check whether it contains the correct email address and public key, i.e., the one requested by LD in 16. The same is true for the CIF, which behaves similarly to the LD. The formal model of BrowserID presented in Appendix D.3 contains these fixes.

Login Injection Attack. During the login process, the origin of the response postMessage (26 in Figure 4.3), which contains the CAP, is not checked. An attacker (e.g., in a malicious advertisement iframe within RP-Doc or in a parent window of RP-Doc), can continuously send postMessages to the RP-Doc with his own CAP in order to log the user into his own account (see [Bug13a]). The outcome of this attack is similar to session swapping. For example, if the attacker is able to log the user into a search engine, the attacker might be able to read the search terms the user enters. This attack violates security w.r.t. session integrity.

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

Identity Forgery (for Primary Mode with Identitiy Bridges only). There are two problems in Sideshow that lead to identity forgery attacks for Gmail addresses; analogously in BigTent with Yahoo email addresses (See [Bug13b] and [Bug13c]).

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

not listed in `openid.signed`). Since Sideshow does not require the email attribute to be in `openid.signed`, it accepts the OpenID assertion, considers the attacker's session to be authenticated for `victim@gmail.com`, and issues UCs for this address to the attacker. This violates security w.r.t. authentication.

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

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

Key Cleanup Failure Attack. When LD creates a key pair (15 in Figure 4.3), it stores the keys in the localStorage (even in ephemeral sessions). When a user quits a session (e.g., by clicking on RP's logout button and closing the browser) the key pair (and the UC) remain in the localStorage, unlike session cookies. Hence, users of shared terminals can read the localStorage (in our model, a `CLOSECORRUPT` allows an attacker to do this) and then, using the key pair and the UC, create valid CAPs to log in at any RP under the identity of the previous user, which violates security w.r.t. authentication. (See [Iss13b] for the discussion in the BrowserID development team.)

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

Cookie Cleanup Failure Attack (Secondary IdP only). The LPO session cookie is not deleted when the browser is closed, even in ephemeral sessions and even if a user logged out at RP beforehand. (In our model, if the attacker issues a `CLOSECORRUPT`, he can therefore still access the LPO session cookie.) Hence, another user of the same browser could request new UCs for *any* identity registered at LPO for that user, and hence, log in at any RP under this identity, which violates security w.r.t. authentication. (See [Iss13a] for the discussion in the BrowserID development team.)

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

4.5.2. Security of the Fixed Systems

We include the fixes mentioned above into our models of the BrowserID primary mode Web system and the BrowserID secondary mode Web system. We then call such Web systems *fixed BrowserID primary mode Web system* and *fixed BrowserID secondary mode Web system*, respec-

tively. We now state theorems that authentication and session integrity as defined in Section 3.4 are fulfilled by both kinds of Web systems and sketch the respective proofs.

Theorem 1. Let $\mathcal{BID}^p$ be a fixed BrowserID primary mode Web system with a network attacker. Then, $\mathcal{BID}^p$ is secure w.r.t. authentication.

To prove Theorem 1, we analyze the request to an honest RP r upon which r returned a service token n for some id i. We show that it must contain a valid CAP (for the identity i). For this, it must in particular contain a valid UC and a matching IA. We show that the UC must have been created by the IdP that governs the identity i (which is honest by assumption). We can then show that only b can request a UC at the IdP for the identity i, and that b does not leak the private key that corresponds to the public key used for this UC, and that this key was chosen from b's set of fresh nonces. Thus, only b can know the key that is used in the creation of the UC in the CAP. We show that neither the private key corresponding to the public key in the UC, nor the IA can leak to the attacker. Thus, the attacker cannot have sent the request that caused r to create the service token n. Also, n does not leak to the attacker. The attacker can therefore not know n, which contradicts the assumption and proves that the authentication property is satisfied. The full proof is contained in Appendix D.4

Before we state our theorem about session integrity of the fixed BrowserID primary mode Web system, recall that BrowserID also supports a non-interactive mode (see automatic CAP creation in Section 4.2): If the user has logged in at some RP before, the user gets automatically logged in again if she visits that RP again. This feature, however, interferes with session integrity as the creation of a service session is not necessarily in the same SSO session as the user's consent actions. We therefore analyze a slightly weaker session integrity property, which we call *indirect session integrity*. This property is the same as session integrity but without the requirement that the actual login must be part of the SSO session, i.e., we consider Definition 13 without Statement iv).[15]

Theorem 2. Let $\mathcal{BID}^p$ be a fixed BrowserID primary mode Web system with a network attacker. Then, $\mathcal{BID}^p$ is secure w.r.t. indirect session integrity.

We start our reasoning from an (arbitrary) processing step Q_{login} that fulfills the predicate $\mathsf{loggedIn}(Q_{\text{login}}, b, r, id, n)$ for some browser b, some RP r, some identity id, and some service token n. We trace back on connected processing steps (steps that are related in a causality chain as defined in Definition 9 in Section 3.3. We show that the user must have interacted

[15] We note that if the non-interactive mode of BrowserID would be removed, our proof shows that we would then have (full) session integrity.

with a login dialog delivered by LPO. Recall that the scripts *script_rp_index*, *script_lpo_cif*, and *script_lpo_ld* are implemented as state machines. We use the transitions of the states of these scripts to derive previous processing steps. We distinguish between two cases: (1) The respective CAP (which was used to authenticate the user) was created in a login dialog and (2) the respective CAP was created by a CIF. In (1), using information stored in localStorage and sessionStorage, we are able to show that the user must have selected the identity to log in at r. As the user only selects identities owned by b in our model, we can show that *id* (which is the identity RP considers to be logged in) is actually owned by b. We further are able to trace back to the start of the SSO session and show that this session was in fact created by b for r and that the user actually selected the identity *id* in this SSO session. (Note that the selection of the IdP is performed in the very same step as for BrowserID the IdP is selected by the selection of the identity). For (2), we show that (based on the values in the localStorage), the user must have completed some login flow as in (1) previously. Hence, the indirect session integrity property is satisfied. The full proof is contained in Appendix D.5

Theorem 3. Let $\mathcal{BID}^s$ be a fixed BrowserID secondary mode Web system with a network attacker. Then, $\mathcal{BID}^s$ is secure w.r.t. authentication.

Again, as for the primary mode above, we prove this Theorem by contradiction. The reasoning is very similar to above. The full proof is contained in Appendix E.3.

Theorem 4. Let $\mathcal{BID}^s$ be a fixed BrowserID secondary mode Web system with a network attacker. Then, $\mathcal{BID}^s$ is secure w.r.t. indirect session integrity.

We also prove this theorem similar to the primary mode. As we only have to consider LPO as the only IdP in this case and the fact that the login dialog is not navigated, the proof is simpler for the secondary mode as our reasoning has to consider fewer steps and fewer parties involved. The full proof can be found in Appendix E.4.

4.5.3. Related Work

As mentioned in the introduction, there are only a few previous approaches to analyze the security of BrowserID. None of these analyses, however, have revealed the vulnerabilities discovered in this thesis.

Bai et al. analyze the security of BrowserID's primary mode in [Bai+13], but focus on the automatic extraction of a model from a protocol implementation. Their tool-based analysis is not very detailed and does not reveal any of the attacks presented in this thesis. The analysis identifies only two rather trivial and implementation specific attacks. In the first attack, if an

RP does not enforce the usage of HTTPS correctly, CAPs might be sent unencrypted and can then replayed by the attacker to an RP. In the second attack, if the RP does not protect itself from XSRF attacks sufficiently, the attacker can force the user to log in under some different identity by instructing the user's browser to send a CAP provided by the attacker to the RP. In our model, RPs follow best practices for the usage of HTTPS and XSRF protection and hence, are not vulnerable to these attacks.

Dietz and Wallach demonstrated a technique to secure BrowserID when specific flaws in TLS are considered that break the confidentiality and integrity of TLS [DW14]. They describe an attack scenario in which the adversary acts as a man-in-the-middle between a browser and an IdP. To protect against this kind of attacker, they recommend to use a variant of TLS client authentication to authenticate users to IdPs. Their work is based on a formal model using BAN logic, that is based on a very high-level model of the BrowserID primary mode and does not take the complexity of the Web infrastructure into account. In our model, we assume that TLS works as intended, i.e., TLS provides confidentiality and integrity, and follow Mozilla's implementation of BrowserID.

In [Han+10; SS13; SKS10; WCW12], potentially problematic usage of postMessages and the OpenID interface are discussed. On a high-level, the problems discussed there are similar to some attacks on BrowserID presented in this thesis. While very useful, these papers do not consider BrowserID or formal models, and they do not formalize security properties for Web applications or establish formal security guarantees.

4.6. Privacy of BrowserID

In this section, we study the privacy guarantees of the BrowserID system with primary IdPs. Regarding privacy, Mozilla states that "...the BrowserID protocol never leaks tracking information back to the Identity Provider." [Bam+18a] and "Unlike other sign-in systems, BrowserID does not leak information back to any server [...] about which sites a user visits." [Mil11].[16] While this statement itself is not a formal definition of the level of privacy that BrowserID is supposed to provide, these and other statements (see, e.g., [Adi11; Bam+18b]) make it certainly clear that, unlike for other SSO systems, IdPs should not be able to learn to which RPs their users log in. This intuition matches our privacy definition presented in Section 3.4.3.

While trying to prove this property for BrowserID, we found severe attacks against the privacy of BrowserID which show that BrowserID does not provide privacy in the presence of a

[16] Clearly, in the current state of BrowserID a malicious LPO server could gather information about users' log in history. However, an integration of the code currently delivered by LPO into the browser, as envisioned, would avoid this issue. Currently, Mozilla's LPO needs to be trusted.

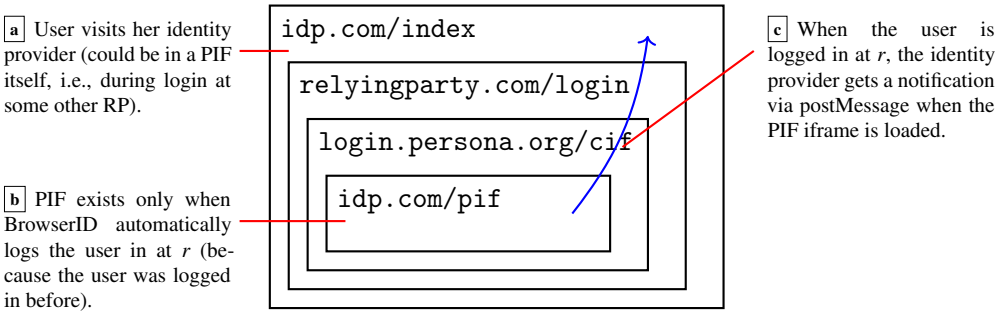


Figure 4.5.: The three main steps of the privacy attack on BrowserID. Using a specially crafted PIF document, a malicious IdP can notify itself via postMessage when the user is logged in at some RP *r*.

malicious IdP. The attacks clearly show that privacy is broken for any reasonable definition of privacy. Unfortunately, our attacks are not caused by a simple implementation error, but rather a fundamental design flaw in the BrowserID protocol. Fixes for this flaw are conceivable, but not without major changes to the design of BrowserID as discussed in Section 4.6.2. Note that for the secondary mode, privacy is not achievable at all as the only IdP (LPO) has access to full information about the login flow. This mode does not target privacy at all.

4.6.1. Privacy Attacks on BrowserID

As explained in Section 3.4.3, for privacy, we consider Web attackers. Consequently, for our attacks to work, it suffices that the IdP is a Web attacker. The attacks even work if all DNS servers, RPs, and LPO are honest, and all parties use encrypted connections. In what follows, we present several variants of attacks on privacy.

PostMessage-Based Attack. The adversary is a malicious IdP that is interested to learn whether a user is logged in at RP *r*. Figure 4.5 illustrates the main steps:

Step a. First, the victim visits her IdP. In BrowserID, email providers serve as IdPs, and therefore it is not unlikely that a user visits this Web site (e.g., for checking email or to use other services). As the IdP usually has some cookie set at the user's browser, it learns the identity of the victim. The IdP now creates a hidden iframe containing the login page of *r*.

Step b. The login page of *r* (now loaded as an iframe within IdP's Web site) includes and runs the BrowserID script. As defined in the BrowserID protocol, the script creates the communication iframe (see "Automatic CAP Creation" in Section 4.2), which in turn checks whether the email address is marked as logged in at *r* in the localStorage of the user's browser. Only then it will try to create a new CAP, for which it needs a PIF (the same as in Phase ii in Figure 4.3).

Step c. The PIF is loaded from the IdP. Note that from this action alone, the IdP does not learn

where the user wants to log in. However, instead of the original (honest) PIF document, the IdP can send a modified one that sends a postMessage to the parent of the parent of the parent of its own window, which in this setting is the IdP document that was opened by the user in Step a. When the IdP receives this message in the document from Step a, it knows that the PIF was loaded, and therefore, that the user is currently logged in at r.

Note that the IdP can repeatedly apply the above as long as the user stays on the IdP's Web site. During this period, the IdP can see whether or not the user is logged in at the targeted RP. Clearly, the IdP can simultaneously run the attack for different RPs in order to track the user's login status for all such RPs. In particular, the IdP can distinguish whether a user is logged in at RP r or r', which violates the privacy property sketched above. In our formal model, the malicious IdP would run the attacker script R^{att} in `idp.com/index` and in `idp.com/pif` (see Figure 4.5) in order to carry out the attack.

Variant 1: Waiting for UC requests. The IdP first acts as in Step a. Now, it could passively wait for incoming requests for the PIF document or UC requests on its server, which tell the IdP that a provisioning flow (probably initiated by Step a) was started. This variant cannot be executed in parallel and is less reliable in practice, though.

Variant 2: PIF as Attack Source. Step a can also be launched from within a PIF itself (i.e., the PIF also takes the role of `idp.com/index` above). This way, while the user logs in at some r_1, the IdP could check whether the user is logged in at r_2, for any r_2.

Variant 3: Scanning the Window Structure (I). Instead of using a postMessage to alert the IdP's outer document about the existence of the inner PIF document, the outer document could as well repeatedly scan the window tree of the iframe containing r's Web site: While the IdP sees almost no information about r's document in the iframe (as it is not same origin), it can see the list of subwindows (i.e., the CIF, and possibly other iframes). For these frames, again, it would see the subwindows, especially the PIF, which it could identify uniquely by checking whether it is same origin with the IdPs outer window.

Variant 4: Scanning the Window Structure (II). In Variant 2, using a same-origin check, the malicious IdP can uniquely identify the PIF in the window structure. This same-origin check could be skipped and it could only be checked whether a PIF is generated, based on the window structure alone. While this is less reliable, this attack could be launched by *any* third party Web attacker (not only the IdP to which the user's email address belongs) to check whether the victim is logged in at r or not.

We verified all variants of the attacks in our model as well as in a real-world BrowserID setup. Implementing proofs-of-concept required only a few lines of (trivial) JavaScript. In most attack

variants, we directly or indirectly use the structure of the windows inside the Web browser as a side channel. To our knowledge, this is the first description of this side channel for breaking privacy in browsers. The attacks have been reported to and confirmed by Mozilla [Bug15].

4.6.2. Fixing the Privacy of BrowserID

Fixing the privacy of BrowserID seems to require a substantial redesign of the system. Regarding the presented attacks, BrowserID's main weakness is the window structure. The most obvious mitigation, modifying the CIF such that it always creates the PIF (even if the user has not logged in before), does not work: To open the PIF, the CIF looks up (in the localStorage) the user's identity at the current RP to derive the address of the PIF. If the user has not logged in before, this information is not available.

Another approach would be to use cross-origin XHRs to replace the features of the PIF. This solution would require a major revision in the inner workings of BrowserID and would not protect against Variant 1.

Such fixes, however, do not overcome a major design problem of BrowserID: The protocol heavily relies on LPO. Strictly speaking, BrowserID is not a decentralized SSO system, as every login always involves LPO as a central authority. The most critical part of the protocol, the creation of the IA and the CAP is performed under the origin of LPO with all details accessible to this party. Further, LPO is hard-coded into the implementation and cannot be replaced or freely determined ad-hoc. Hence, if LPO is considered dishonest, privacy (as well as security and session integrity) is trivially broken.

4.7. Limitations of the Analysis

Besides the general limitations of our approach as discussed in Section 2.8, our model of BrowserID is also subject to some limitations, which we discuss below.

We treat the primary and the secondary mode of BrowserID as two different protocols in two separate analyses. In principle, a protocol that allows for several modes in parallel, an attacker could try to mix up two or more modes to create new attacks. Also, new problems can arise from the ability of a domain owner to switch between primary and secondary mode over time. Recall that LPO decides which mode to be used based on the presence of the support document, which is also cached at LPO. Hence, a full analysis of both modes in combination, which introduces even more complexity, might reveal new attacks. BrowserID, however, has been decommissioned and is nowadays only interesting as a research example. In addition, the secondary mode was planned to be only a temporary extension of BrowserID [Mil11]. Further, as we see in Section 4.6.1 below,

BrowserID fails to achieve its privacy goal. Therefore, we move on in Chapter 5 and design an SSO system that actually fulfills all security and privacy goals, rather than combining the analyses of both modes of an SSO protocol that does not meet one of its main goals.

Identity bridges, as described in this chapter, have not been included into the formal analysis as they, from the BrowserID perspective, act as a normal IdP. A full formal analysis of these services would necessarily include a formal analysis of all SSO protocols that are bridged over to BrowserID, which clearly goes beyond the scope of an analysis of BrowserID itself.

5. Design and Analysis of SPRESSO

As we have seen in the previous chapter, the design of a secure SSO system is non-trivial and attacks are very easy to overlook. In this chapter, we present SPRESSO, an SSO protocol that is inspired by BrowserID's unique design goals, in particular privacy, but is designed from scratch. We use the WIM to design and model this protocol right from the start. Based on this formal model, we perform a rigorous analysis of SPRESSO regarding the authentication, session integrity, and privacy properties presented in Section 3.4. Our analysis shows that SPRESSO indeed meets these properties making SPRESSO the first SSO system for the Web that provides strong security and privacy guarantees. We provide a prototypical implementation of SPRESSO at [FKS19].

We created SPRESSO to be a decentralized, open system. In SPRESSO, users are identified by their email addresses, and email providers certify the users' authenticity. Compared to OpenID 2.0 [FR+07], users do not need to learn a new, complicated identifier — our approach is similar to BrowserID in this respect. But unlike in BrowserID, there is no central authority in SPRESSO (see also the discussion in Section 5.2). 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.

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 no support extensions or plug-ins.

In this chapter, we first provide a detailed description of SPRESSO in Section 5.1 and discuss this protocol in Section 5.2. We continue by describing the formal model of SPRESSO in Section 5.3. We then show that SPRESSO is a sound SSO protocol that provides the necessary authentication and session integrity properties in Section 5.4. In Section 5.5, we present our result that SPRESSO indeed provides privacy. We conclude with our proof-of-concept implementation of SPRESSO in Section 5.6.

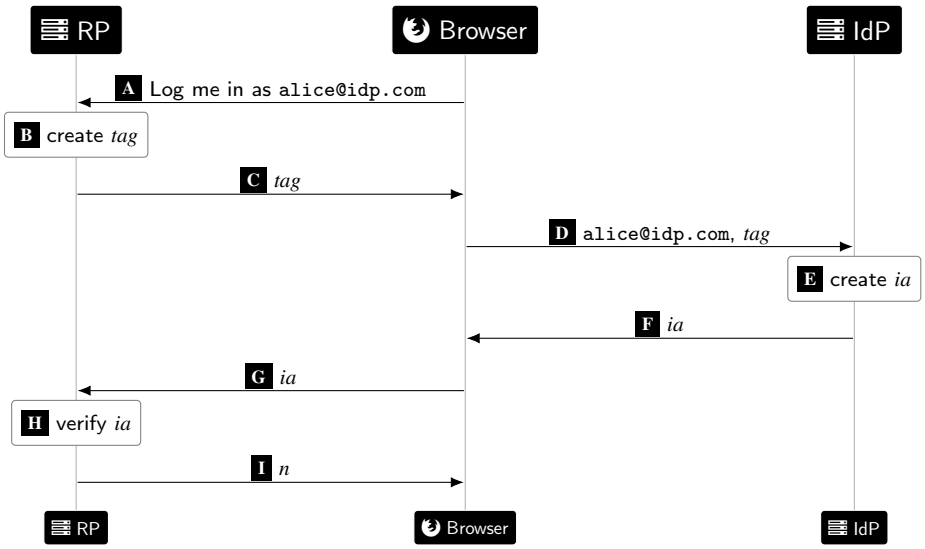


Figure 5.1.: High-level idea of SPRESSO.

5.1. Protocol

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.

5.1.1. Overview

On a high level, the login flow consists of the following steps (see also Figure 5.1: First, on the RP Web site, the user starts the login flow by entering her email address [A]. RP then creates what we call a *tag* by encrypting its own domain name and a nonce with a freshly generated symmetric key [B] and sends the tag to the user [C]. The user forwards this tag along with her email address to the IdP [D]. 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)* [E]. The IA is then sent back to the user's browser [F] who forwards it to the RP [G]. RP checks the signature and consistency of the data signed [H] and then considers the user with the given email address to be logged in (by creating a

service session and sending the service token to the user 1).

We note that passing the IA to the RP is done using a forwarder within the browser (the RP determines which one is used). The task of this forwarder is to deliver the IA 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. We will discuss the forwarder mechanism in detail below.

5.1.2. 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 5.2. We use the names RP, IdP, and FWD for the servers of the respective parties. We use RPdoc, 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, 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 (containing 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`[1] 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 the notation introduced in Chapter 2, we denote this term by

$$tag := \mathsf{enc_s}(\langle \texttt{RPDomain}, rpNonce \rangle, tagKey)\,.$$

RP further selects (a domain of) an FWD (e.g., a fixed one from its settings, see below). Now, RP stores *tag*, *iaKey*, the FWD domain, and the email address in its session data store under the session key *loginSessionToken* and sends *tag*, *tagKey*, *FWDDomain*, and *loginSessionToken* as response to the POST request by RPdoc 8.

RPdoc now opens the login dialog in 9. This window contains the login dialog from IdP (IdPdoc) so that the user can log in to IdP (if not logged in already). In this step, special care must be taken not to reveal the identity of the RP to the IdP via the Referer header (see the discussion

[1] We use fixed "well-known" paths at IdPs as proposed in [RFC5785].

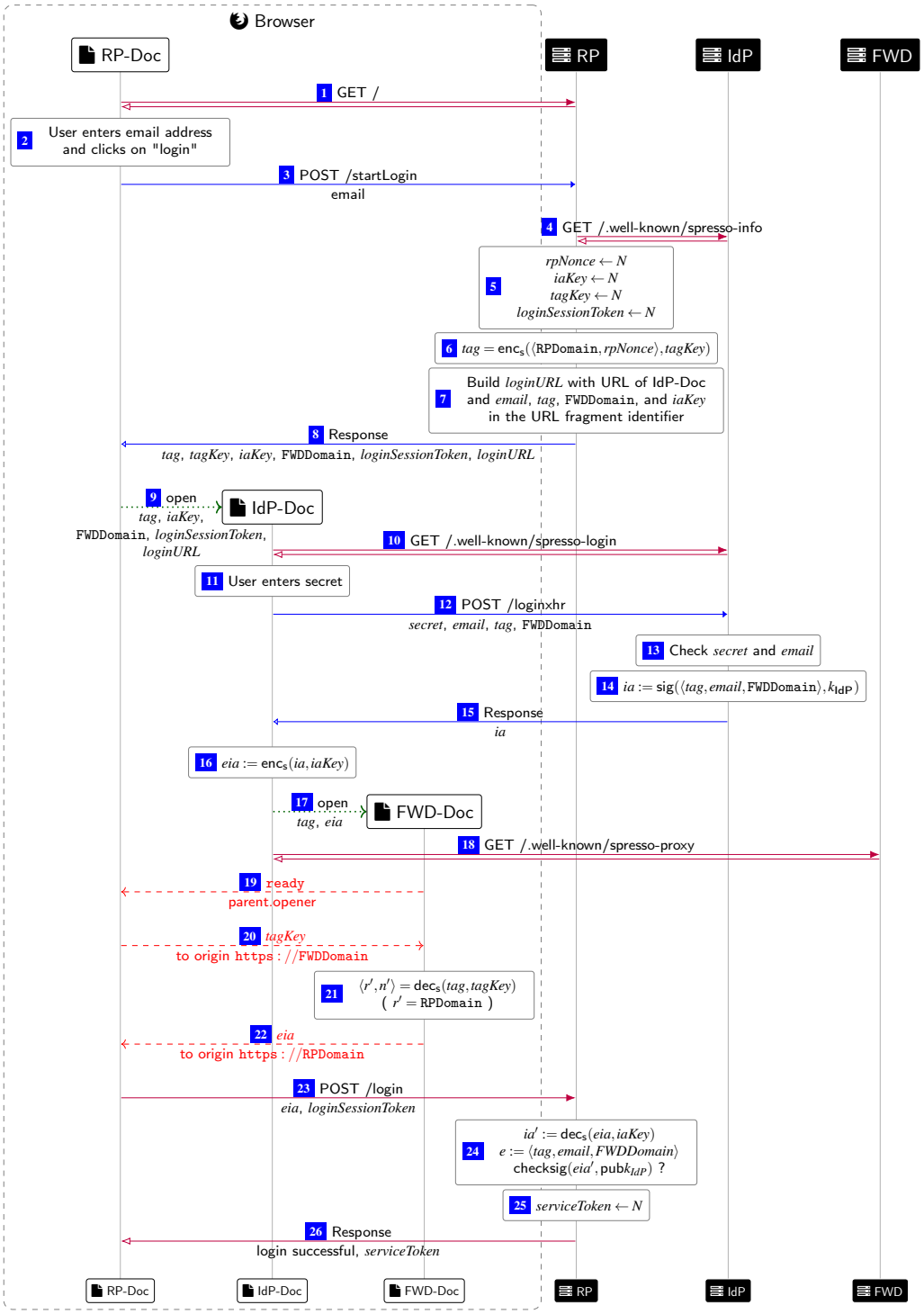


Figure 5.2.: Overview on the SPRESSO login flow. (See Page 11 for notation.)

in Section 5.2). In the URL of the login dialog, RP includes the user's email address, the tag, the FWD domain, and the iaKey.[2]

After the browser loaded IdPdoc from IdP, the user enters her password[3] matching her email address 11. The password, the email address, the tag, and the FWD domain are now sent to IdP 12. After IdP verified the user credentials 13, it creates the identity assertion as the signature

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

using its private signing key k_{IdP} 14 and then returns *ia* to IdPdoc 15. 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 5.2), IdPdoc now encrypts the IA using the iaKey 16:

$$eia := \mathsf{enc_s}(ia, iaKey)\ .$$

Then, IdPdoc opens an iframe with the URL of FWDdoc, passing the tag and the encrypted IA to FWDdoc. After the iframe is loaded 18, FWDdoc sends a postMessage to its parent's opener window, which is RPdoc 19. This postMessage with the sole content "ready" triggers RPdoc to send the *tagKey* to FWDdoc, where in the postMessage the origin of FWD with HTTPS is declared to be the only allowed receiver of this message 20. FWDdoc uses the key to decrypt the tag and thereby learns the intended receiver (RP) of the IA 21. As its last action, FWD forwards the encrypted IA *eia* via postMessage to RPdoc (using RP's HTTPS origin as the only allowed receiver) 22.

RPdoc receives *eia* and sends it along with the *loginSessionToken* to RP 23. RP then decrypts *eia*, retrieves ia' and checks whether ia' is a valid signature for $\langle tag, email, FWDDomain\rangle$ under the verification key $\mathsf{pub}(k_{IdP})$ of the IdP, where *tag*, *email*, and *FWDDomain* are taken from the session data identified by *loginSessionToken* 24.

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 a service token (as explained in Section 3.1) 25 and sends this token to the browser 26.

[2] This 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.

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

5.2. Discussion of the Protocol

Before we present our formal proof of SPRESSO's authentication and privacy properties in the subsequent sections, we first provide more intuition and motivation. We first informally discuss some challenges, (selected) potential attacks and what measures we took when designing and implementing SPRESSO to prevent such attacks. These attacks also illustrate the complexity and difficulty of designing a secure and privacy-respecting Web-based SSO system. We further discuss the necessity of the forwarder and how this mechanism can be improved by future Web standards. 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, RPa, 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 RPa 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 RPb using the user's email address. The attacker's RP could then create a tag of the form $\mathsf{enc}_{\mathsf{s}}(\langle \mathtt{RPb}, \mathit{rpNonce} \rangle, \mathit{tagKey})$ using the domain of an honest RP RPb, instead of RPa. 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 RPb (and obtain the service token from RPb), and thus, log in at RPb 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, RPb), which in the attack above is not the domain of the document loaded in the attacker's RP window (RPa). 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 RPa, before Step 22. 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 is, for example, the identity injection attack on BrowserID presented in Section 4.5.1. 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 24. 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.[4] It could further try to corellate 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 Section 5.5).

Recall that 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.

Therefore, for our system to provide authentication and privacy, we require that FWDs behave honestly. Below we discuss technical means 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 15 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.

Enforcing Honest FWDs. Before we discuss existing and upcoming technologies to enforce

[4] If 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.

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 note that the IdP needs to trust the (script of the) forwarder to not leak the IA to a malicious party. If the forwarder fails to do so, this compromises the authentication property for all of the IdP's users. Still, the IdP can always deny using a specific, untrustworthy forwarder. Hence, both parties, the RP and the IdP have to agree on a forwarder which they both trust.

Third, 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.[5] Therefore, SPRESSO does not provide FWD with information to track at which RP a specific user logs in.[6]

Now, as for *enforcing* honest FWDs, first note that an honest FWD server is supposed to always deliver the same fixed document containing a 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.

Current browsers have two shortcomings which prevent a removal of the forwarder: The postMessage mechanism allows a sender to only specify the origin that a receiver must have (otherwise, the browser does not deliver the message). Hence, the sender does not have any guarantee about which script will actually process the message on the receiver's side. Another shortcoming is that a browser does not provide any guarantees about the content of an iframe (except that the content is initially loaded from the URL provided by the creator of the iframe).

Using current technology, a user could use a browser extension which can be very simple: This extension has to make sure that in fact only the correct 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 browsers do not support extensions on all platforms. The first solution (i.e., the extension checks only that correct JavaScript is loaded) would at least

[5] We 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.

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

still work for users on such platforms, albeit with reduced security and privacy guarantees.

A better approach would be an improvement of browsers: If the postMessage mechanism is extended such that the sender is able to specify the actual script or the whole document (i.e., by providing a hash of the full document) of the receiving window, RP-Doc does not need to rely on the forwarder to actually deliver the correct document. This extension of postMessages would allow RP-Doc to send the postMessage in Step 26 (Figure 5.2) containing *tagKey* to the correct forwarder script. In this postMessage, RP-Doc would not need to require the origin of receiver to be the forwarder (our proof relies on the fact that the correct forwarder script receives *tagKey*, see also later). Instead, the forwarder script could also be provided by the IdP. As *tagKey* does not get stored in the scriptstate and is only processed by the (trusted) forwarder script, *tagKey* does not leak to the IdP, still protecting the users' privacy. Obviously, also the authentication property still holds true in this (slightly) changed setting. This extension of the postMessage mechanism would be sufficient to remove the forwarder from SPRESSO.

Without this extension of postMessages, a future version the Subresource Integrity (SRI) standard [Akh+16] would at least enable IdP-Doc to enforce the forwarder to deliver the correct forwarder script. While the initial SRI draft [Akh+14] proposed a way to enforce that an iframe's document must match a certain hash value, this feature was removed in later versions of SRI. The developers of SRI are still considering to implement this feature (see discussion at [Iss15] as well as the note to include iframes in future versions in the current SRI specification [Akh+16]). Using SRI would still not allow us to remove the forwarder, but enable us to distrust the forwarder in our analyses of the authentication and session integrity properties.

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.

The *Referrer Policy* proposal [ES17] introduces a way to suppress the Referer header for specific Web pages. Referrer policies allow to specify several conditions under which a browser must suppress or shorten the Referer header.[7] A Web server can deliver a referrer policy as an HTTP header or as part of HTML code. Major browsers already support the Referrer Policy proposal [Moz19]. In SPRESSO, RPs always set the Referrer Policy to suppress the Referer header in every HTTP(S) response.

In an earlier version of SPRESSO presented in [FKS15b], we proposed a slightly different

[7] In a nutshell, the Referrer policy can completely forbid the browser to send the Referrer header, strip the URL contained in the Referer header to its origin, allow the Referer header to be sent unaltered, or use a combination of these modes depending on whether the Referer header will be sent cross-origin or over a non-HTTPS connection.

solution. At that time, the Referrer Policy proposal had not been made, but a similar way to suppress the Referer header had already been introduced with HTML5: a special attribute for links, which causes the Referer header to be suppressed (`rel="noreferrer"`).[8] 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 19 is sent to the opener window of IdPdoc. To preserve the opener handle while at the same time hiding the Referer header, we first opened the new window with a redirector document loaded from RP and then navigate this window to IdPdoc (using a link with the noreferrer attribute set and triggered by JavaScript). This causes the Referer header to be cleared as well, while the opener handle is preserved.[9] Using this redirect solution, however, is more complicated and therefore, has been simplified in SPRESSO as presented here. Our formal analysis (presented in this document and in [FKS15b]) implies that with both 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 [RFC1035], 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

[8] Note that the specification of this attribute has also been incorporated into the Referrer Policy specification [ES17].

[9] Another 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.

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

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 a classical email-based account creation process 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.

Comparison with BrowserID. As discussed earlier in this thesis, BrowserID was the first and so far only SSO system designed to provide privacy. Nonetheless, as discussed in Section 4.6, we discovered severe attacks on BrowserID 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. Also, BrowserID has the disadvantage that it relies on a single trusted server (LPO). Furthermore, LPO is quite complex, involved into several server interactions in every login process, and most importantly, by design, gets full information about the login behavior of users.

In SPRESSO, we do not have a central authority comparable to LPO. SPRESSO, however, requires a forwarder, which is responsible for transferring an identity assertion from the IdP to the RP. In a login process, however, the FWD server is not a fixed entity and can be freely chosen. The FWD server needs to provide only a fixed single and very simple JavaScript and 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.

Finally, BrowserID is a rather complex SSO system (with at least 64 network and inter-frame messages in a typical login flow[11] compared to only 19 in SPRESSO). This complexity implies that security vulnerability go unnoticed more easily. During our formal analysis of BrowserID, we have found several severe attacks breaking authentication and privacy claims.

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 browers) 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.

5.3. Formal Model

We here present the formal model of SPRESSO, which is the basis for our formal analysis of authentication, session integrity, and privacy. Our model closely follows the description of SPRESSO presented in Section 5.1.2. As this model is also based on the template for SSO protocols presented in Chapter 3, our description of this model presented here — on a high-level — looks similar to our description of the model of BrowserID (Section 4.4). Obviously, the details of the two models differ significantly.

We model SPRESSO as an SSO Web system (in the sense of Section 3.1). We call $\mathcal{SWS} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ an *SPRESSO Web system* if it is of the form as described in Appendix F.1. In

[11] Counting HTTP request and responses as well as postMessages, leaving out any user requests for GUI elements or other non-essential resources.

what follows, we outline the components of such a Web system.

Following Definition 4, we define the system $\mathcal{W}^{sp}$ to be composed of the (pairwise disjoint) subsets Web, Net, B, RP, IDP, DNS, and Other. The set Other contains forwarders. For readability, we refer to this set by FWD, i.e., FWD = Other.

As described in Section 3.4, for analysis of authentication and session integrity, we use an *SPRESSO Web system with a network attacker* that is an SPRESSO Web system with $\mathsf{Web} = \emptyset$ and $\mathsf{Net} = \{\mathsf{attacker}\}$. We also write $\mathcal{SWS}^{auth}$ if we refer to an SPRESSO Web system with a network attacker. For the analysis of privacy, we define an *SPRESSO Web system for privacy analysis* that follows Definition 16 (see also Section 5.5 below).

The set $\mathcal{N}$ of nonces is partitioned into four (disjoint) sets: The (infinite) set $N^{\mathcal{W}^{sp}}$ contains the nonces that are available for each DY process in $\mathcal{W}^{sp}$ ($N^{\mathcal{W}^{sp}}$ is set as a sequence in the initial configuration of the system), the set K_{TLS} contains the keys that will be used for TLS encryption, the set K_{sign} contains the keys that will be used by IdPs for signing IAs (signing keys),[12] and the set $\mathsf{Secrets} \subseteq \mathcal{N}$ is the set of passwords (secrets) the browsers share with the identity providers. The nonces in K_{TLS}, K_{sign}, and Secrets are distributed according to their purpose over the initial states of the processes (see below).

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 one IP address each. As usual, 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 (we denote the assigned by dom). Browsers (in B) and DNS servers (in DNS) do not have a domain. Each domain is assigned a unique private key from K_{TLS} and for each domain, the "owner" (according to dom^{-1}) is given the respective private key in its initial state. For each of these keys, the public key (i.e., $\mathsf{pub}(k)$ for $k \in K_{\mathrm{TLS}}$) is given to (the initial state of) browsers, RPs, and attackers in the form of a dictionary indexed by the key's domain.

(User) identities are used as specified in Definition 1, which also matches the nature of email addresses. Each browser $b \in \mathsf{B}$ is assigned a (finite) set of identities and their respective secrets (out of Secrets). Identities and secrets are unique to a browser, i.e., browsers have disjoint sets of identities and disjoint sets of secrets.

Similar to our model of BrowserID, we provide a list of pairs of identities and corresponding secrets in the initial state of their governor and set the initial state of browsers such that browsers provide secrets to scripts that run under a secure origin of their identities' governor.

[12] As in the model for BrowserID, we call the public counterpart of a (private) signing key ($k \in K_{\mathrm{sign}}$) a verification key ($\mathsf{pub}(k)$).

The set $\mathcal{S}$ contains four scripts, with their string representations defined by script: besides the attacker script R^{att}, the set contains the honest scripts that describe the honest Web pages of RPs, IdPs, and the forwarders (see below for more details).

The set E^0 contains only the trigger events as specified in Section 2.3.

Recall that in an SPRESSO login flow, RP issues a *tag*, which in turn is placed by the IdP in an *identity assertion*. We first define these elements formally as follows:

Definition 19. A *tag* is a term of the form $\mathsf{enc_s}(\langle d, n\rangle, k)$ for some domain d, a nonce $n \in \mathcal{N}$, and a nonce (here used as a symmetric key) k. ◇

Definition 20. An *identity assertion (IA)* is a term of the form $\mathsf{sig}(\langle t, e, d'\rangle, k)$ with t being a tag, e an identity, d' a domain, and k a nonce. We call a term *eia* an *encrypted identity assertion (EIA)* if *eia* is of the form $\mathsf{enc_s}(ia, k')$ with *ia* being an IA and k' a nonce.

◇

The (signing) keys (nonces) for IAs are distributed in the initial states of IdPs, i.e., each IdP is given one (private) key that the IdP uses to sign all IAs for identities it governs.

We now sketch the processes and the scripts in $\mathcal{W}^{sp}$ and $\mathcal{S}$ (see Appendix F.1 for full details). As mentioned, our modeling closely follows the description in Section 5.1.2.

Browsers. Browsers (in B) are defined as described in Section 2.5 and configured (in their initial state) similar as in the model of BrowserID: In their initial state, browsers are given a list of their identities, secrets for these identities (stored in a dictionary under a secure origin of the respective governor), and a dictionary of public keys used for TLS.

See Appendix F.1.5 for further details on browsers in $\mathcal{W}^{sp}$.

IdPs. Each IdP (in IDP) is a Web server. It responds requests to three distinct paths (see below), where it serves the login dialog Web page (`script_idp`), the support document containing its public key, and 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 (Web) sessions. Only users that are authenticated by such a session (or provide credentials to create such a session) can receive IAs from the IdP.

In their initial state, each IdP is given its (private) TLS keys, the private signing key that is used to sign IAs, and information about the identities the IdP governs. For the "user database" of an IdP, we use the same idea as in BrowserID (see Section 4.4.1) and provide the IdP in its initial state a dictionary that maps secrets to a list of identities the secret is valid for.

Initially, IdPs are honest. When receiving a corrupt message IdPs become corrupted. Similar to the definition of corruption for the browser, IdPs then start sending out all messages that are derivable from their state. This means, that they effectively become Web attackers.

(Web) sessions at IdPs. IdPs maintain (Web) sessions to track whether a user has authenticated before. These sessions are tracked using cookies.

HTTPSRequests to IdPs. IdPs reply to certain requests only. All of these requests have to be received over HTTPS. We describe each kind of HTTPS request below:

`GET /.well-known/spresso-login`. An IdP replies to such an request with a response containing the script `script_idp`. Into the initial state of this script, the IdP encodes whether the browser is already logged in or not. Further, the IdP issues an XSRF token to the browser (in the same way RPs do).

`GET /.well-known/spresso-info`. If an IdP receives such a request, it replies with its (public) verification key that can be used to verify its identity assertion signatures.

`POST /sign`. Using a POST request to this path requests an IA. Such a request must contain an identity and a valid session id for that identity or a password for that identity. Further, such a request must contain a tag and a forwarder domain. When processing such a request, the IdP creates an identity assertion for these values and replies (in an HTTPS response) with that identity assertion. Further, the IdP always creates a new session for the identity (with a cookie in its reply).

RPs. RPs follow our description above. They serve the script `script_rp` and accept requests for starting and completing an SPRESSO login flow (see below). Note that all HTTPS responses an RP sends out contain the `ReferrerPolicy` header that disables the Referer header in Web browsers for all requests caused by scripts in documents created by such a response. Initially RPs are honest and can be corrupted similar to IdPs.

In its initial state, each RP is given the (private) TLS key for its domain, a dictionary of public TLS keys, and the domain of a forwarder that they use for SPRESSO.

During a run, an RP learns (public) verification keys of IdPs (the RP fetches this key from the respective IdP via HTTPS). The RP maintains a dictionary for these public keys. In this dictionary, the RP stores each key it learned under the domain of the respective IdP. If the RP already knows the (public) verification key for an IdP domain, the RP uses this key right away to verify the signature of IAs for this domain.

RPs maintain two kinds of (Web) sessions: *login sessions*, which are only used during the login phase of a user, and service sessions (as described in Section 3.1).

Login sessions at RPs. A login session is created by an RP to track the login flow of a user. The login session is identified by a login session token that is passed to the user when starting a login. RP records, for each login session, the id of the user, the tag used for the login flow (containing a fresh nonce), and the key that is used to encrypt the identity assertion.

Service sessions at RPs. Service sessions at RPs are defined as usual, i.e., an RP issues a service token that refers to an entry for the service session in RP's state.

HTTPSRequests to RPs. RPs reply to certain requests only. All of these requests have to be received over HTTPS.

We describe each kind of HTTPS request below:

`GET /` and `POST /`. An RP replies to all requests to the path `/` by providing the script *script_rp*.

`POST /startLogin`. If an RP receives such a request, the RP starts a login flow. The RP expects[13] that such a request contains a user identity in its body. If the RP does not know the domain of this id (i.e., there is no public key for verifying signatures of the domain's IdP in RP's state), the RP first retrieves this information by sending an HTTPS request to that domain. If the RP has retrieved that information or already knows the domain of this id, the RP creates a set of fresh keys, *tagKey* and *iaKey*. It creates a tag that contains the RP's domain and a fresh nonce and that is encrypted (symmetrically) using *tagKey*. Further, RP creates a login session (see above) and answers the request with (among others) the tag, the login session token, the domain of the forwarder that RP selected,[14] and the URL of the login dialog at IdP.[15] This URL contains (as parameters) the tag, the identity, the forwarder domain, and *iaKey*.

`POST /login`. If an RP receives such a request, the RP expects this request to contain several values: a login session token that refers to a current login session, and an identity assertion (encrypted with the respective key stored in the login session). This identity assertion must contain (according to the login session) the correct tag, the correct identity, and the correct forwarder domain and the identity assertion must have a valid signature by the IdP of the identity. If these conditions are met, the RP creates a service session and responds with an HTTPS response containing the service token.

See Appendix F.1.7 for more details on RPs.

[13] When we write *expect*, we mean that the RP tries to decompose the message into these values and if this is not possible, the RP stops the processing step without further action.

[14] In our model for RPs, the forwarder is statically configured in RP's state. For privacy analysis, we consider that all (honest) RPs have configured the same forwarder.

[15] The URL of the login dialog is derived from the domain of the identity.

FWDs. Forwarders (in FWD) are Web servers that have only one state (i.e., their relation does not modify the state) and serve only the script `script_fwd` (for all HTTPS requests). In their initial state, forwarders are only given their private TLS keys to receive HTTPS requests.

Forwarders cannot become corrupted. As discussed in Section 5.2, if a dishonest forwarder is used in an SPRESSO flow, attacks are possible. In our analysis, however, we capture all possible runs of an SPRESSO SSO Web system. In such a run, the attacker can easily simulate an honest RP that uses a dishonest forwarder (an attacker process). Honest browsers are not restricted in interacting with this (simulated) RP and can also log in there.

See Appendix F.1.8 for further details.

DNS Servers. Each DNS server (in DNS) contains the assignment of domain names to IP addresses and answers DNS requests accordingly.

SPRESSO scripts. The set $\mathcal{S}$ of the Web system $\mathcal{SWS}$ consists of the scripts R^{att}, *script_rp*, *script_idp*, and *script_rp*.

The script R^{att} is the attacker script (see Section 2.5) and the formal modeling of the remaining (honest) scripts follows the descriptions below (see Appendix F.1.10 for more details). Similar to our model for BrowserID, we model these scripts as state machines that use the subterm q in their scriptstate to track their state.

Relying party script (script_rp). The script *script_rp* defines the script of RP's Web page. In this script, the user chooses an identity and then this script takes care of invoking the SPRESSO login flow.

When run, the script behaves as follows:

$q \equiv$ `start` This is the initial state. In this state, the user selects an identity and the script sends an XHR to RP (determined by the script's origin). The script transitions to the state `expectStartLoginResponse`.

$q \equiv$ `expectStartLoginResponse` The script continues if it has received a response to the XHR sent above. It stores information received in that XHR in its scriptstate (e.g., the login session token, *tagKey*, and the forwarder domain). The script transitions to the state `expectFWDReady` and opens the login dialog with the URL received in the XHR. Recall that this URL is supposed to contain the tag, the identity, the forwarder domain, and *iaKey*.

$q \equiv$ `expectFWDReady` In this state, the script continues if it has received a postMessage from the forwarder iframe located in the login dialog the script opened earlier. This postMessage tells the script that the forwarder is ready. The script then answers this postMessage with

a postMessage to the forwarder iframe that contains *tagKey* and transitions to the state `expectEIA`.

$q \equiv$ `expectEIA` The script checks whether it has received a new postMessage from the forwarder iframe (located in the login dialog as above). The script expects that this postMessage contains an encrypted identity assertion, which the script forwards along with the login session token (from the scriptstate) to its RP in an XHR. The script then transitions to the state `expectServiceToken`.

$q \equiv$ `expectServiceToken` In this state, the script does nothing.

Identity provider script (script_idp). This script represents the login dialog of IdPs. This script authenticates to an IdP, forward a tag to that IdP, expect an identity assertion in return and then pass this identity assertion (encrypted) to a forwarder iframe.

The states of this script and its behaviour are as follows:

$q \equiv$ `start`. This is the initial state. The script expects an identity, a tag, and a forwarder domain in the URL parameters of its document. The script sends an XHR to its origin containing the identity, the secret (password) for this origin, the tag, and the forwarder domain. Then, the script transitions to the state `expectIA`.

$q \equiv$ `expectIA`. In this state, the script continues if it has received a response to the XHR sent above and expects that this XHR response contains an identity assertion. The script encrypts that identity assertion with the *iaKey* taken from its document's URL and creates an iframe witht he URL of the forwarder. It passes the tag and the encrypted identity assertion to the forwarder iframe as URL parameters. Finally, the script transitions to the state `stop`.

$q \equiv$ `stop`. In this state, the script does nothing.

Forwarder script (script_fwd). The script *script_fwd* is responsible for the forwarder iframe. The main purpose of this script is to forward an identity assertion to a document that runs under the origin the assertion is issued for.

The script can take three states:

$q \equiv$ `start`. This is the initial state. The script sends a postMessage to the opener of the login dialog (the login dialog is supposed to be the parent of the window containing this script). This postMessage indicates that the forwarder script is ready. The script then transitions to the state `expectTagKey`.

$q \equiv$ `expectTagKey`. The script continues if it has received a postMessage from the same window as above. The script expects that this postMessage contains *tagKey*, which it uses to decrypt the tag. Recall that the tag contains the domain of the RP that issued the tag. The script now sends a postMessage to the same window as above. This postMessage contains the encrypted identity assertion (which the script takes from its document's URL) and is restricted to be delivered only to the domain of the RP (contained in the tag). The script then transitions to the final state `stop`.

$q \equiv$ `stop`. In this state, the script does nothing.

5.4. Authentication and Session Integrity of SPRESSO

We now show that SPRESSO satisfies authentication and session integrity. For both properties, we use the SPRESSO SSO Web model with a network attacker as described in the previous section. Recall that this system contains, beside a 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 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 first prove that SPRESSO satisfies authentication:

Theorem 5. Let $\mathcal{SWS}^{auth}$ be an SPRESSO Web system as defined above. Then $\mathcal{SWS}^{auth}$ is secure w.r.t. authentication.

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 containing an IA has to be received by the RP.

We then use these properties and the general Web system properties shown in Section 2.9 to prove the theorem. We assume that the authentication property is not satisfied and lead this to a contradiction. In particular, we show that an attacker must get hold of a valid identity assertion created by an honest IdP for some honest user and for some honest RP. We show that such an IA is only ever issued to a browser that owns the respective identity. In the browser, the only script that can receive the issued IA is *script_idp* (running under an origin of the IdP). This script only ever passes the IA to a forwarder iframe that is loaded from an honest forwarder (the forwarder determined by the RP that requested the IA). We show that, in this iframe, only the

script *script_fwd* can be loaded and that this script only ever forwards the IA to a document that is loaded from a (secure) origin of the RP the IA was issued for. Such a document must contain the script *script_rp* that only ever forwards the IA to the RP it was loaded from. As the RP never leaks any IA forwarded by this script, we have that the IA cannot be known by the attacker (or any other corrupted party). As the RP only ever sends out a service token to the sender of such an IA (using HTTPS), the attacker cannot get hold of a service token for an honest user.

The full proof is provided in Appendix F.2.

Next, we prove that SPRESSO also provides session integrity:

Theorem 6. Let $\mathcal{SWS}^{auth}$ be an SPRESSO Web system as defined above. Then $\mathcal{SWS}^{auth}$ is secure w.r.t. session integrity.

For session integrity, we show that for every (network) event in which an honest browser gets logged in under some identity at an honest RP (i.e., the browser receives a service token for this identity at the RP), the browser must have sent a corresponding (HTTPS) request that contains a login session token that is bound to the same user identity at the RP. Such a request must have been sent by the script *script_rp* as this is the only script that is delivered by the RP and hence, the only script that can instruct the browser to send requests to the RP that pass the `Origin` header check at the RP. This script only ever uses a login session token that must have been set when a certain HTTPS response was processed (by the browser). This response must be a response to a request sent by the script that starts the login flow at the RP. This request is constructed (by the script) such that it only ever requests a login at the RP for an identity owned by the browser.[16] When the RP processes this request, it creates the login session token and binds this token to the identity in the request. Hence, the user only gets logged in for identities she owns and the steps in which the user starts a login flow, selects an IdP, selects an identity, and gets finally logged in are all part of the same SSO session.

Note that the design of SPRESSO makes this proof very straightforward as the first three steps (start of the login flow, selection of the IdP, and selection of the identity) are all the very same processing step. Hence, it is sufficient to show that this processing step and the processing step in which the user gets logged in, are part of the same SSO session and the identity selected by the user in the first step matches the one for which the user gets logged in. As SPRESSO only uses scriptstates to store the login session token in the browser, several (different) login flows cannot be mixed up.

We provide the full proof in Appendix F.3.

[16] Recall that we assume that a user only enters identities actually owned by this user.

5.5. Privacy of SPRESSO

For the analysis of privacy, we consider SPRESSO Web systems for privacy analysis following the Definition 16 in Section 3.4.3. These Web systems are derived from "normal" SPRESSO Web systems described in Section 5.3. In such a system for privacy analysis, we have one or more Web attackers, no network attackers, one honest DNS server, one honest forwarder, one (challenge) browser, and two honest relying parties r_1 and r_2 (with domains dr_1 and dr_2, respectively). 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 attackers subsume also potentially dishonest forwarders, DNS servers, relying parties, and other servers. We use a challenge browser as the only honest browser (see Section 3.4.3). The honest relying parties are set up such that they already contain the (public) verification keys for each domain registered at the DNS server, modeling that the RPs have "warm" caches or are able to retrieve these keys through an anonymous channel as discussed in Section 5.2.

We denote Web systems as described above by $\mathcal{SWS}^{priv}(dr)$, where dr is the domain of the relying party given to the challenge browser in this system. See Appendix F.1.12 for a full formal definition.

We now show that SPRESSO indeed enjoys privacy, i.e., that no IdP can tell at which RP a user is about to log in or has logged in.

Theorem 7. Every SPRESSO Web system for privacy analysis is IdP-private.

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.

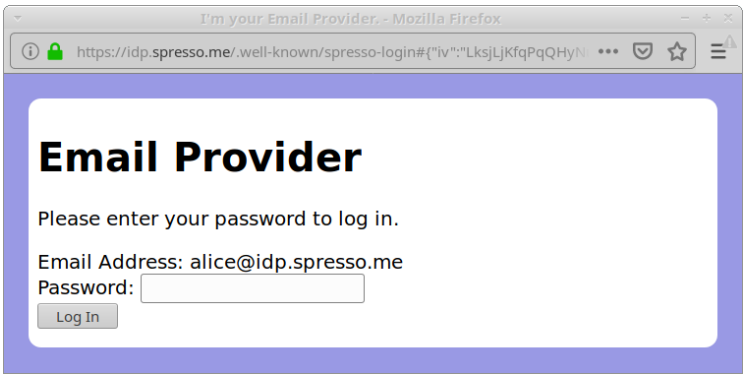
The full proof is provided in Appendix F.4.

5.6. Implementation

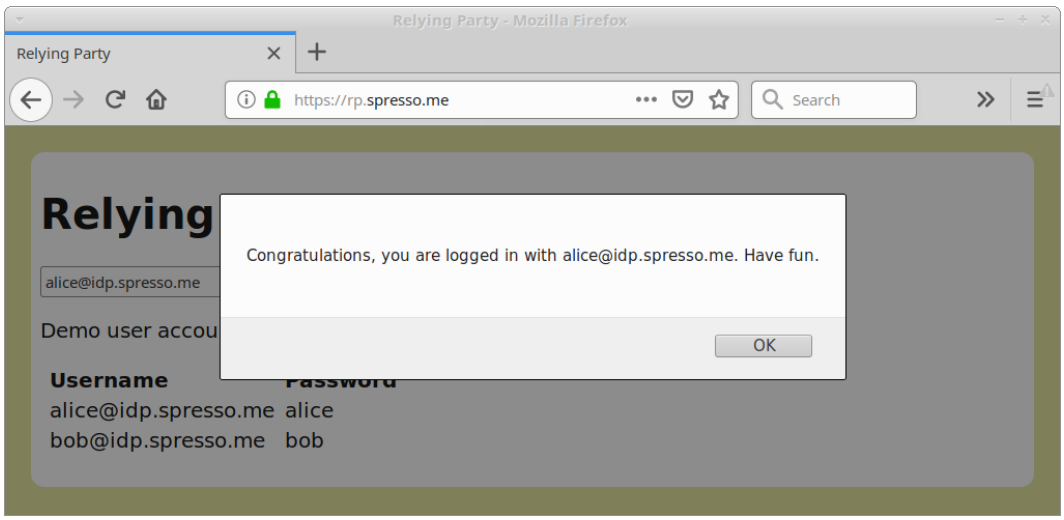
We developed a proof-of-concept implementation of SPRESSO in about 700 lines of JavaScript and HTML code. It contains all presented features of SPRESSO itself and a typical IdP. Figure 5.3 shows screenshots of our implementation. The source code and a demo site are available at [FKS19]. Our implementation closely follows the model presented in Section 5.3.

The three servers (RP, IdP and FWD) are written in JavaScript and are based on node.js and its built-in *crypto* API. We provide these servers as Docker containers for easy deployment. 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.

(a) The relying party asks the user to log in.



(b) The user's email provider (IdP) prompts the user for her password.



(c) The user is logged in at the RP.

Figure 5.3.: Screenshots of our prototypical SPRESSO implementation.

6. Conclusion and Future Work

Our work is the first to systematically analyze privacy in the Web in a formal way. As privacy itself is worthless without proper security, our approach includes the analysis of security properties as well. We not only show shortcomings of a real-world SSO system, BrowserID, but also, for the first time, use formal methods to design a secure and private SSO system from scratch.

Our approach is based on the WIM, the most comprehensive model of the Web infrastructure to date. The WIM captures many vital aspects of the inherently complex Web. Most importantly, the WIM includes a detailed model of modern Web browsers with many basic and advanced features starting at the handling of multiple windows up to Web messaging and Web storage. A detailed model like the WIM is essential for security and privacy analysis as even inconspicuous details such as the window structure in a browser can break privacy — as demonstrated by the attacks on the privacy of BrowserID that we present in this thesis.

To perform our analyses on this solid foundation, we defined generic properties for security and privacy: The security properties are — as usual in Dolev-Yao style models — based on conditions that every single run of a system has to meet, e.g., that certain nonces are not derivable by an adversary. Privacy, however, requires a more complex definition. This property captures the inability of an adversary to distinguish between different choices of a user. Hence, the notion of derivability of some secret is not powerful enough as even subtle behavioral differences may provide hints to the attacker. Therefore, we have based our privacy property on the notion of static equivalence and compare the attacker's view on two separate runs. Every pair of runs that are compared in this way are equally induced by a schedule chosen by the adversary and only differ by the user's choice at one specific step determined by the attacker. For the privacy property to be fulfilled, every possible pair of such runs must match static equivalence of the attacker's view, i.e., its runs are effectively indistinguishable for the attacker. Obviously, this kind of analysis based on the comparison of two similar runs induced by the adversary is particularly challenging compared to the analysis of the security properties that only reason on possible runs individually.

To enable privacy analysis, we have extended and improved the WIM. In particular, we had to remove technical non-determinism in many places, where this non-determinism was in conflict with our notion of similar runs needed for privacy. Further, we introduced a way for the attacker

to schedule and to have (limited) control over such runs and also added challenge browsers that allow an attacker to trigger a user choice which by itself is opaque to the adversary.

To facilitate future analyses of Web SSO systems, we generalized security and privacy properties such that they can be used more easily in the analysis of other Web SSO systems. This way, the results of such analyses are also comparable to each other. Although only applied to Web SSO systems so far, these properties can be adapted for analyses of other Web protocols, standards, or applications.

As the first application of the WIM itself, we have analyzed BrowserID, the first Web SSO system aimed at providing privacy to its users. During our analysis, we found severe attacks on BrowserID. These attacks do not only impact BrowserID's security, but also break BrowserID's privacy property. While we were able to fix all security relevant findings and showed that the fixed version of BrowserID satisfies the security goals of such an SSO system, we conclude that regarding privacy, BrowserID is broken beyond repair. We have reported all of our findings to Mozilla, who acknowledged these severe problems and awarded us several bug bounties.

Our analysis of BrowserID also shows that modeling of a protocol in a comprehensive framework of the Web infrastructure, such as the WIM, already provides some insight. For example, while analyzing the source code of BrowserID to create the model of BrowserID, we discovered attacks on BrowserID's identity bridge feature. Obviously, the creation of a model in such a concise framework as the WIM does not replace a formal proof, but allows an analyst to discover some problems already in an early stage of the analysis.

Inspired by BrowserID's unique privacy goal and driven by the question whether it is possible to create a privacy-preserving Web SSO, we have designed and analyzed SPRESSO. Our analysis shows that SPRESSO indeed provides strong privacy and security guarantees. This makes SPRESSO the first Web SSO system to provide privacy and also the first Web SSO system to feature a formal proof for its privacy and security properties from the beginning.

SPRESSO is designed to be very easy to use. The protocol is based on native Web features only and does not require any third party add-ons or extensions which might complicate adoption or might break compatibility in the future.

Similar to BrowserID, in SPRESSO, email providers become IdPs, and users use their email addresses to log in. SPRESSO can be seamlessly integrated into RPs, who often already use email addresses as login names. When a user enters her email address at an RP, an SPRESSO login flow can be automatically started without confusing the user with additional separate login options. Thanks to SPRESSO's strong privacy, this automatic use of SPRESSO does not impose any disadvantage on users in contrast to other Web SSO protocols. Further, as email addresses are already a key factor in the traditional approach for account registration and recovery, an IdP that wants to break into its users' accounts at RPs maliciously also does not gain more power

than before.

SPRESSO allows RPs to provide SSO to its users without the usual privacy implications. Email providers can advertise SPRESSO support as a new feature for their users. SPRESSO's advantages can be very interesting for email providers who are already committed to privacy today.

As SPRESSO is designed as an open eco-system, any party can start to support SPRESSO right now. In contrast to BrowserID, SPRESSO is fully decentralized, and there exists no fixed, single central authority, which can always control any aspect of the protocol. This makes SPRESSO also suitable to be used not only in the open Web but also in closed, local systems, such as companies' internal intranets.

We have implemented SPRESSO in an entirely usable proof-of-concept implementation. To ease wide adoption, we plan to provide libraries for popular Web application frameworks and development languages.

Future versions of SPRESSO can, besides strong privacy and security, also include several other features, such as additional user claims (i.e., attributes, such as name or postal address) to be provided and certified by the IdP to the RP. Obviously, such features need to be carefully selected as some features known from other protocols such as OAuth 2.0, e.g., granting access to users' resources stored at IdPs to RPs, break privacy.

So far, all security and privacy proofs based on the WIM have been carried out manually without any tool support. A next step to assist future work based on the WIM is to mechanize this framework and its proofs. Such mechanization provides analysts with several advantages: Proofs can be automatically checked for correctness, which helps to avoid manual errors and can also facilitate the creation of a (new) formal proof by pointing the analyst to next steps. Proofs and lemmas can also be reused more easily in subsequent analyses with the tool ensuring that these proofs can be applied. This automatic re-verification is also helpful to re-check old proofs in updated versions of the WIM in order to verify that new WIM features do not break old results. Another advantage of such mechanization is that the accessibility of models and the analyses is improved for non-experts. Having a verified mechanized model also enables the generation of executable code for real-world implementations from this model that matches its verified logic. As a long term goal, we envision further automation, including the automatic extraction of a model from real-world source code or even specifications. Our vision is to support the whole development process for Web standards and protocols with formal analysis from the drawing board up to deployment. Our manual approach already has drawn the attention of standardization bodies, such as the IETF and the OpenID Foundation (see also our work on OAuth 2.0 [FKS16a] and OpenID Connect 1.0 [FKS17b]). Hence, a tool enabling members of such organizations to conduct formal analyses would be very valuable.

A promising candidate for a tool foundation to mechanize WIM analyses is F* [F*; Swa+13]. F* is a functional programming language aimed at program verification. Properties of a program, such as security properties, but also functional properties, can be stated precisely using F*'s comprehensive type system. F* translates all type constraints into satisfiability modulo theory (SMT) problems. SMT problems are decision problems based on first-order logic (see, e.g., [MDS07]). There exist several tools that aim at automatically solving SMT problems. F* uses the SMT solver Z3 [MB08]. F* has been already applied successfully to analyze and verify complex cryptographic protocols, such as TLS 1.2 and TLS 1.3 [Bha+17].

Independent of mechanization, the WIM can also be further extended. For example, the WIM browser can be extended with full support for Cross-Origin Resource Sharing [WHA19a], including the so-called pre-flight requests. These pre-flight requests enable browsers to check whether a cross-origin request has given clearance to be sent out. Another possible extension is support for security extensions of DNS, such as *DNS Security (DNSSEC)* [RFC4033] and *DNS-based Authentication of Named Entities (DANE)* [RFC6394]. DNSSEC introduces a PKI for DNS that is used to authenticate information stored in DNS. DANE builds upon DNSSEC to allow DNSSEC's PKI to be extended to applications outside of DNS, such as HTTPS.

We are also interested in the analysis of other properties not covered so far. One attractive example of such a property in the context of Web SSO is unlinkability of user logins across different RPs. On a high-level, this property is similar to the privacy property analyzed in this thesis but is based on the assumption that RPs instead of IdPs are malicious, collude with each other, and try to correlate user activities. Some Web SSO protocols, such as OpenID Connect with so-called pairwise subject identifiers [Sak+14], promise to provide this kind of unlinkability and can serve as case studies.

The current version of SPRESSO does not provide unlinkability w.r.t. malicious RPs. Redesigning SPRESSO such that it provides both, privacy w.r.t. malicious IdPs and unlinkability w.r.t. malicious RPs is also an interesting but challenging goal. Clearly, having both properties at the same time would make SPRESSO an even more attractive Web SSO protocol.

As we have seen in this thesis, formal treatment of security and privacy properties in the Web infrastructure is an exciting topic with substantial real-world impact. We have paved the way for formal analyses of this field and demonstrated that our formal approach is suitable to be applied in the complex environment of the Web. We hope that this approach will find wide adoption to ensure a high level of security as well as privacy in the future.

A. The Web Infrastructure Model (WIM)

Here, we provide technical definitions that complete our description of the WIM in Chapter 2.

A.1. Terms and Notations

Definition 21 (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 $\pi_1(\mathsf{dec}_{\mathsf{a}}(\mathsf{enc}_{\mathsf{a}}(\langle \mathsf{a}, \mathsf{b} \rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 22 (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 $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc}_{\mathsf{a}}(\langle a, b, c \rangle, \mathsf{pub}(k))$ is interpreted to be the message $\langle a, b, c \rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

Definition 23 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m \rangle$, for a, $f \in \mathsf{IPs}$ 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 $\mathcal{M}^{\nu}$ are called *protoevents* and are denoted $\mathcal{E}^{\nu}$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^{\nu}\langle\rangle}$, 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.). ◇

Definition 24 (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 2.2. For a term t, we denote its normal form as $t{\downarrow}$. ◇

Definition 25 (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 $t \sim pattern$. For a sequence of patterns *patterns* we write $t \dot{\sim} patterns$ to denote that t matches at least one pattern in *patterns*.

For a term t' we write $t'|pattern$ 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 $\langle \top, 42 \rangle \sim p$, $\langle \bot, 42 \rangle \not\sim p$, and

$$\langle \langle \bot, \top \rangle, \langle \top, 23 \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \top, \bot \rangle \rangle | \, p = \langle \langle \top, 23 \rangle, \langle \top, \bot \rangle \rangle \, .$$

Definition 26 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$, and $t_1, \ldots, t_n \in \mathcal{T}_N$.

By $\tau[t_1/x_1, \ldots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \ldots, n\}$. ◇

Definition 27 (Sequence Notations). For a sequence $t = \langle t_1, \ldots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \ldots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \ldots, t_n, y \rangle$. For a sequence $r = \langle r_1, \ldots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \ldots, t_n, r_1, \ldots, r_m \rangle$. For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.) ◇

Definition 28. A *dictionary over X and Y* is a term of the form

$$\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. 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 A.1 shows the short notation for dictionary operations. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. (Note that if a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply non-deterministically to one of both elements.) If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]\,[k_i] = v_i \tag{A.1}$$

$$[k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \ldots, k_n : v_n] - k_i = [k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \ldots, k_n : v_n] \tag{A.2}$$

Figure A.1.: Dictionary operators with $1 \leq i \leq n$.

Given a term $t = \langle t_1, \ldots, t_n \rangle$, 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 29. A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ 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 $\overline{p} = \langle 3, 1 \rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

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 $o.\texttt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Definition 30 (Concatenation of terms and sequences). For a term $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = (b_1, b_2, \ldots)$, we define the *concatenation* as $a \cdot b := (a_1, \ldots, a_i, b_1, b_2, \ldots)$.

◇

Definition 31 (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 occurence of that element in X is removed. ◇

A.2. 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 presented in the following.

A.2.1. URLs

Definition 32. A *URL* is a term of the form

$$\langle \texttt{URL}, protocol, host, path, parameters, fragment \rangle$$

with $protocol \in \{\texttt{P}, \texttt{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), a domain $host \in \mathsf{Doms}$, $path \in \mathbb{S}$, $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, and $fragment \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs. ◇

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$. We sometimes also write $\mathsf{URL}_{path}^{host}$ to denote the URL $\langle \texttt{URL}, \texttt{S}, host, path, \langle\rangle, \bot \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 29):

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

A.2.2. Origins

Definition 33. An *origin* is a term of the form $\langle host, protocol \rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\texttt{P}, \texttt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins. ◇

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

A.2.3. Cookies

Definition 34. A *cookie* is a term of the form $\langle name, content \rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle value, secure, session, httpOnly \rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, *secure*, *session*, *httpOnly* $\in \{\top, \bot\}$. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^{V}$ 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 HTTP(S) responses, only the components *name* and *value* are transferred as a pairing of the form $\langle name, value \rangle$.

A.2.4. HTTP Messages

Definition 35. An *HTTP request* is a term of the form shown in (A.3). An *HTTP response* is a term of the form shown in (A.4).

$$\langle \texttt{HTTPReq}, \mathit{nonce}, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle \tag{A.3}$$

$$\langle \texttt{HTTPResp}, \mathit{nonce}, \mathit{status}, \mathit{headers}, \mathit{body} \rangle \tag{A.4}$$

The components are defined as follows:

- $\mathit{nonce} \in \mathcal{N}$ serves to map each response to the corresponding request
- $\mathit{method} \in \mathsf{Methods}$ is one of the HTTP methods.
- $\mathit{host} \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $\mathit{path} \in \mathbb{S}$ is a string indicating the requested resource at the server side
- $\mathit{status} \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard)
- $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters
- $\mathit{headers} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, containing request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \texttt{Origin}, o \rangle$ where o is an origin,
  - $\langle \texttt{Set-Cookie}, c \rangle$ where c is a sequence of cookies,
  - $\langle \texttt{Cookie}, c \rangle$ where $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \texttt{Location}, l \rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \texttt{Referer}, r \rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \texttt{Strict-Transport-Security}, \top \rangle$,
  - $\langle \texttt{Authorization}, \langle \mathit{username}, \mathit{password} \rangle \rangle$ where $\mathit{username}, \mathit{password} \in \mathbb{S}$,
  - $\langle \texttt{ReferrerPolicy}, p \rangle$ where $p \in \{\texttt{noreferrer}, \texttt{origin}\}$.
- $\mathit{body} \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

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

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, /\texttt{show}, \langle\langle \texttt{index}, 1\rangle\rangle, [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S}\rangle], \langle \texttt{foo}, \texttt{bar}\rangle\rangle \tag{A.5}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x\rangle\rangle \tag{A.6}$$

An HTTP `POST` request for the URL `http://example.com/show?index=1` is shown in (A.5), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar}\rangle$. A possible response is shown in (A.6), which contains an httpOnly cookie with name `SID` and value n_2 as well as the string representation `somescript` of the script $\mathsf{script}^{-1}(\texttt{somescript})$ (which should be an element of $\mathcal{S}$) and its initial state x.

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

Definition 36. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k'\rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively. ◇

We say that an HTTP(S) response matches or corresponds to an HTTP(S) request if both terms contain the same nonce.

Example 7.

$$\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{A.7}$$

$$\mathsf{enc_s}(s, k') \tag{A.8}$$

The term (A.7) shows an encrypted request (with r as in (A.5)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (A.8) is a response (with s as in (A.6)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (A.7).

A.2.5. DNS Messages

Definition 37. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, \mathit{domain}, \mathit{nonce}\rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$. ◇

Definition 38. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, domain, result, nonce \rangle$ with $domain \in \mathsf{Doms}$, $result \in \mathsf{IPs}$, $nonce \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$. ◇

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.

A.3. Atomic Processes, Systems and Runs

Entities that take part in a network are modeled as atomic processes. An atomic process takes a term that describes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

Definition 39 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

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

where $I^p \subseteq \mathsf{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^{\mathcal{V}}\langle\rangle} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (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 $(\eta_1, \eta_2, \dots)$ 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 40 (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 sequence[1] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$. ◇

Definition 41 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

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

where

1. (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
2. $e_{\text{in}} = \langle a, f, m \rangle \in E$ is an event,

[1] Here: Not in the sense of terms as defined earlier.

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

with

1. $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
2. $E_{\text{out}} = E^{\nu}_{\text{out}}[m_1/v_1, \ldots, m_i/v_i]$
3. $S'(p) = s^{\nu}[m_1/v_1, \ldots, m_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$
4. $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$
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 occurences 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 42 (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 $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ 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 containing infinite trigger events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

A.4. 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 43 (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 $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V. ◇

For example, the term a can be derived from the set of terms $\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\}$, i.e., $a \in d_{\{\}}(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

Definition 44 (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 for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_\mathcal{N}$, $s' \in \mathcal{T}_\mathcal{N}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that $E, s' \in d_{V_{\text{process}}}(\{e, s\})$. ◇

A.5. Attackers

The so-called *attacker process* is a Dolev-Yao process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties (using corrupt messages).

Definition 45 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* 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 $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_0, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$. ◇

Note that in a Web system, we distinguish between two kinds of attacker processes: Web attackers and network attackers. Both kinds match the definition above, but differ in the set of

assigned addresses in the context of a Web system. While for Web attackers, the set of addresses I^p is disjoint from other Web attackers and honest processes, i.e., Web attackers participate in the network as any other party, the set of addresses I^p of a network attacker is not restricted. Hence, a network attacker can intercept events addressed to any party as well as spoof all addresses. Note that one network attacker subsumes any number of Web attackers as well as any number of network attackers.

A.6. Browsers

Following the informal description of the browser model in Section 2, we now present the formal model of browsers.

A.6.1. Scripts

Recall that a *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). See Algorithm F.7 on Page 265 for an annotated example of a script.

Definition 46 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts. ◇

Definition 47 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$. ◇

A script is called by the browser which provides it with state information (such as the script's last scriptstate and limited information about the browser's state) s. The script then outputs a term s', which represents the new scriptstate and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \dots$ 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, the so-called *attacker script* outputs everything that is derivable from the input.

attacker script R^{att}:

Definition 48 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$. ◇

A.6.2. Web Browser State

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

Definition 49. A *window* is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$ Documents (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ 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 $w.\texttt{activedocument}$ 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., $b.\texttt{opener} = a.\texttt{nonce}$.

Definition 50. A *document* d is a term of the form

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

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

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

Definition 51. For two window terms w and w' we write

$$w \xrightarrow{\text{childof}} w'$$

if $w \in^{\langle\rangle} w'.\texttt{activedocument.subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure and we write $\xrightarrow{\text{childof}^*}$ for the reflexive transitive closure. ◇

In the Web browser state, HTTP(S) messages are tracked using *references*. These are defined as a pairing of a an identifier for the type of the request (*XHR* for XMLHTTPRequests, or *REQ* for normal HTTP(S) requests) and a nonce (which never leaves the browser unless the browser becomes corrupted).

We can now define the set of states of Web browsers. Note that we use the dictionary notation that we introduced in Definition 28.

Definition 52. The *set of states $Z_{webbrowser}$ of a Web browser atomic Dolev-Yao process* consists of the terms of the form

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

with the subterms as follows:

- $windows \subset^{\langle\rangle} \mathsf{Windows}$ contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).
- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ contains a list of secrets that are associated with certain origins (i.e., passwords of the user of the browser at certain Web sites). Note that this structure allows to have a single secret under an origin or a list of secrets under an origin.
- *cookies* is a dictionary over Doms and sequences of Cookies modeling cookies that are stored for specific domains.
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle \mid o \in \mathsf{Origins}, r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ maps domains to TLS encryption keys.
- $sts \subset^{\langle\rangle} \mathsf{Doms}$ stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in \mathsf{IPs}$ defines the IP address of the DNS server.

- *pendingDNS* $\in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url \rangle$. In these pairings, *reference* is an HTTP(S) request reference (as above), *request* contains the HTTP(S) message that awaits DNS resolution, and *url* contains the URL of said HTTP request. The pairings in *pendingDNS* are indexed by the DNS request/response nonce.

- *pendingRequests* $\in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f \rangle$ with *reference*, *request*, and *url* as in *pendingDNS*, *key* is the symmetric encryption key if HTTPS is used or $\perp$ otherwise, and f is the IP address of the server to which the request was sent.

- *isCorrupted* $\in \{\perp, \texttt{FULLCORRUPT}, \texttt{CLOSECORRUPT}\}$ specifies the corruption level of the browser.

In corrupted browsers, certain subterms are used in different ways (e.g., *pendingRequests* is used to store all observed messages). ◇

A.6.3. Web Browser Relation

We will now define the relation $R_{\text{webbrowser}}$ of a standard HTTP browser. We first introduce some notations and then describe the functions that are used for defining the browser main algorithm. We then define the browser relation.

Helper Functions. In the following description of the Web browser relation $R_{\text{webbrowser}}$ we use the helper functions Subwindows, Docs, Clean, CookieMerge and AddCookie.

Subwindows. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[2] to windows in the window list $s.\texttt{windows}$, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 53. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

[2] Recall the definition of a pointer in Definition 29.

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

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

Clean. 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 54. Let s be a browser state and d a document. By $\mathsf{Clean}(s,d)$ we denote the term that equals $s.\texttt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) 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, and (3) the values of the subterms `headers` for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.) ◇

CookieMerge. 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 55. For a sequence of cookies (with pairwise different names) *oldcookies* and a sequence of cookies *newcookies*, the set $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\texttt{name} \equiv c'.\texttt{name}$, 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 $c_{\text{old}} \in^{\langle\rangle}$ *oldcookies*, $c_{\text{new}} \in^{\langle\rangle}$ *newcookies*, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is m. ◇

AddCookie. 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 56. For a sequence of cookies (with pairwise different names) *oldcookies* and a cookie c, the sequence $\mathsf{AddCookie}(\mathit{oldcookies}, c)$ is defined by the following algorithm: Let

$m :=$ *oldcookies*. Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m. ◇

NavigableWindows. The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [Ber+14], Section 5.1.4 for this definition.

Definition 57. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

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

◇

Notations for Functions and Algorithms. We use the following notations to describe the browser algorithms:

Non-deterministic chosing and iteration. 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 write, for example,

let x, y **such that** $\langle \texttt{Constant}, x, y \rangle \equiv t$ **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 $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y \rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

Placeholder	Usage
v_1	Algorithm A.9, new window nonces
v_2	Algorithm A.9, new HTTP request nonce
v_3	Algorithm A.9, lookup key for pending HTTP requests entry
v_4	Algorithm A.7, new HTTP request nonce (multiple lines)
v_5	Algorithm A.7, new subwindow nonce
v_6	Algorithm A.8, new HTTP request nonce
v_7	Algorithm A.8, new document nonce
v_8	Algorithm A.4, lookup key for pending DNS entry
v_9	Algorithm A.1, new window nonce
$v_{10}, \ldots$	Algorithm A.7, replacement for placeholders in script output

Table A.1.: List of placeholders used in browser algorithms.

Algorithm A.1: Web Browser Model: Determine window for navigation.

```
1:  function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s′)
2:      if window ≡ _BLANK then    → Open a new window when _BLANK is used
3:          if noreferrer ≡ ⊤ then
4:              let w′ := ⟨v9, ⟨⟩, ⊥⟩
5:          else
6:              let w′ := ⟨v9, ⟨⟩, s′.w̄.nonce⟩
7:          end if
8:          let s′.windows := s′.windows +^⟨⟩ w′
              ↪ and let w̄′ be a pointer to this new element in s′
9:          return w̄′
10:     end if
11:     let w̄′ ← NavigableWindows(w̄, s′) such that s′.w̄′.nonce ≡ window
          ↪ if possible; otherwise return w̄
12:     return w̄′
13: end function
```

Stop without output. We write **stop** (without further parameters) to denote that there is no output and no change in the state.

Placeholders. In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 21). Table A.1 shows a list of all placeholders used.

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 function GETNAVIGABLEWINDOW (Algorithm A.1) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window

Algorithm A.2: Web Browser Model: Determine same-origin window.

function GETWINDOW($\overline{w}$, *window*, s')
  let $\overline{w}' \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}'.\texttt{nonce} \equiv window$ **if possible; otherwise return** $\overline{w}$
  if $s'.\overline{w}'.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ **then**
    return $\overline{w}'$
  end if
  return $\overline{w}$
end function

Algorithm A.3: Web Browser Model: Cancel pending requests for given window.

function CANCELNAV(*reference*, s')
  remove all $\langle reference, req, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$ **for any** *req*, *key*, f
  remove all $\langle x, \langle reference, message, url\rangle\rangle$ **from** $s'.\texttt{pendingDNS}$ **for any** x, *message*, *url*
  return s'
end function

Algorithm A.4: Web Browser Model: Prepare headers, do DNS resolution, save message.

function HTTP_SEND(*reference*, *message*, *url*, *origin*, *referrer*, *referrerPolicy*, s')
  if $message.\texttt{host} \in^{\langle\rangle} s'.\texttt{sts}$ **then**
    let $url.\texttt{protocol} := \texttt{S}$
  end if
  let $cookies := \langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}[message.\texttt{host}] \wedge (c.\texttt{content.secure} \implies (url.\texttt{protocol} = \texttt{S}))\}\rangle$
  let $message.\texttt{headers}[\texttt{Cookie}] := cookies$
  if $origin \not\equiv \bot$ **then**
    let $message.\texttt{headers}[\texttt{Origin}] := origin$
  end if
  if $referrerPolicy \equiv \texttt{no-referrer}$ **then**
    let $referrer := \bot$
  end if
  if $referrer \not\equiv \bot$ **then**
    if $referrerPolicy \equiv \texttt{origin}$ **then**
      let $referrer := \langle \texttt{URL}, referrer.\texttt{protocol}, referrer.\texttt{host}, /, \langle\rangle, \bot\rangle$
        → Referrer stripped down to origin.
    end if
    let $referrer.\texttt{fragment} := \bot$
      → Browsers do not send fragment identifiers in the Referer header.
    let $message.\texttt{headers}[\texttt{Referer}] := referrer$
  end if
  let $s'.\texttt{pendingDNS}[v_8] := \langle reference, message, url\rangle$
  stop $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, v_8\rangle\rangle, s'$
end function

Algorithm A.5: Web Browser Model: Navigate a window backward.

```
1: function NAVBACK(w̄′, s′)
2:     if ∃j̄ ∈ ℕ, j̄ > 1 such that s′.w̄′.documents.j̄.active ≡ ⊤ then
3:         let s′.w̄′.documents.j̄.active := ⊥
4:         let s′.w̄′.documents.(j̄ − 1).active := ⊤
5:         let s′ := CANCELNAV(s′.w̄′.nonce, s′)
6:     end if
7: end function
```

Algorithm A.6: Web Browser Model: Navigate a window forward.

```
1: function NAVFORWARD(w̄′, s′)
2:     if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
           ↪ ∧ s′.w̄′.documents.(j̄ + 1) ∈ Documents then
3:         let s′.w̄′.documents.j̄.active := ⊥
4:         let s′.w̄′.documents.(j̄ + 1).active := ⊤
5:         let s′ := CANCELNAV(s′.w̄′.nonce, s′)
6:     end if
7: end function
```

reference (nonce) *window*, this function 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 $\overline{w'}$ 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, $\overline{w}$ is returned instead (the script navigates its own window).

- The function GETWINDOW (Algorithm A.2) 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.
- The function CANCELNAV (Algorithm A.3) 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.
- The function HTTP_SEND (Algorithm A.4) 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*

Algorithm A.7: Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
  let *tree* := Clean(s', $s'.\overline{d}$)
  let *cookies* := $\langle \{ \langle c.\texttt{name}, c.\texttt{content.value} \rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
    ↪ $\wedge c.\texttt{content.httpOnly} = \bot$
    ↪ $\wedge \left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\} \rangle$
  let *tlw* ← s'.windows **such that** *tlw* is the top-level window containing $\overline{d}$
  let *sessionStorage* := $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
  let *localStorage* := $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let *secrets* := $s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $R \leftarrow \text{script}^{-1}(s'.\overline{d}.\texttt{script})$
  let *in* := $\langle$*tree*, $s'.\overline{d}.\texttt{nonce}$, $s'.\overline{d}.\texttt{scriptstate}$, $s'.\overline{d}.\texttt{scriptinputs}$, *cookies*,
    ↪ *localStorage*, *sessionStorage*, $s'.\texttt{ids}$, *secrets*$\rangle$
  let *state*′ ← $\mathcal{T}_{\mathcal{N}}(V)$, *cookies*′ ← Cookies$^{\mathsf{v}}$, *localStorage*′ ← $\mathcal{T}_{\mathcal{N}}(V)$, *sessionStorage*′ ← $\mathcal{T}_{\mathcal{N}}(V)$,
    ↪ *command* ← $\mathcal{T}_{\mathcal{N}}(V)$, out^{λ} := $\langle$*state*′, *cookies*′, *localStorage*′, *sessionStorage*′, *command*$\rangle$
    ↪ **such that** $(in, out^{\lambda}) \in R$
  let *out* := $out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \ldots]$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$:=
    ↪ $\langle$CookieMerge($s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$, *cookies*′)$\rangle$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$:= *localStorage*′
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$:= *sessionStorage*′
  let $s'.\overline{d}.\texttt{scriptstate}$:= *state*′
  let *referrer* := $s'.\overline{d}.\texttt{location}$
  let *referrerPolicy* := $s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
  let *reference* := $\langle \texttt{REQ}, s'.\overline{w}'.\texttt{nonce} \rangle$
  let *docorigin* := $s'.\overline{d}.\texttt{origin}$
  switch *command* **do**
    case $\langle \texttt{HREF}, url, hrefwindow, noreferrer \rangle$
      let $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *hrefwindow*, *noreferrer*, s')
      let *req* := $\langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle \rangle$
      if *noreferrer* $\equiv \top$ **then**
        let *referrerPolicy* := noreferrer
      end if
      let s' := CANCELNAV(*reference*, s')
      call HTTP_SEND(*reference*, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, s')
    case $\langle \texttt{IFRAME}, url, window \rangle$
      if *window* $\equiv$ _SELF **then**
        let $\overline{w}'$:= $\overline{w}$
      else
        let $\overline{w}'$:= GETWINDOW($\overline{w}$, *window*, s')
      end if
      let *req* := $\langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle \rangle$
      let w' := $\langle \nu_5, \langle\rangle, \bot \rangle$
      let $s'.\overline{w}'.\texttt{activedocument.subwindows}$:= $s'.\overline{w}'.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
      call HTTP_SEND(ν_5, *req*, *url*, $\bot$, *referrer*, *referrerPolicy*, s')

This algorithm is continued on the next page.

```
39:     case ⟨FORM, url, method, data, hrefwindow⟩
40:         if method ∉ {GET, POST} then [3]
41:             stop
42:         end if
43:         let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
44:         if method = GET then
45:             let body := ⟨⟩
46:             let parameters := data
47:             let origin := ⊥
48:         else
49:             let body := data
50:             let parameters := url.parameters
51:             let origin := docorigin
52:         end if
53:         let req := ⟨HTTPReq, v4, method, url.host, url.path, parameters, ⟨⟩, body⟩
54:         let s′ := CANCELNAV(reference, s′)
55:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s′)
56:     case ⟨SETSCRIPT, window, script⟩
57:         let w̄′ := GETWINDOW(w̄, window, s′)
58:         let s′.w̄′.activedocument.script := script
59:         stop ⟨⟩, s′
60:     case ⟨SETSCRIPTSTATE, window, scriptstate⟩
61:         let w̄′ := GETWINDOW(w̄, window, s′)
62:         let s′.w̄′.activedocument.scriptstate := scriptstate
63:         stop ⟨⟩, s′
64:     case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
65:         if method ∈ {CONNECT, TRACE, TRACK} ∧ xhrreference ∉ {N, ⊥} then
66:             stop
67:         end if
68:         if url.host ≢ docorigin.host ∨ url ≢ docorigin.protocol then
69:             stop
70:         end if
71:         if method ∈ {GET, HEAD} then
72:             let data := ⟨⟩
73:             let origin := ⊥
74:         else
75:             let origin := docorigin
76:         end if
77:         let req := ⟨HTTPReq, v4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
78:         let reference := ⟨XHR, s′.d̄.nonce, xhrreference⟩
79:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s′)
```

This algorithm is continued on the next page.

```
80:         case ⟨BACK, window⟩ [4]
81:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
82:             NAVBACK(w̄′, s′)
83:             stop ⟨⟩, s′
84:         case ⟨FORWARD, window⟩
85:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
86:             NAVFORWARD(w̄′, s′)
87:             stop ⟨⟩, s′
88:         case ⟨CLOSE, window⟩
89:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
90:             remove s′.w̄′ from the sequence containing it
91:             stop ⟨⟩, s′
92:         case ⟨POSTMESSAGE, window, message, origin⟩
93:             let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
94:             if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
                ↪ ∧(origin ≢ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
95:                 let s′.w̄′.documents.j̄.scriptinputs := s′.w̄′.documents.j̄.scriptinputs
                    ↪ +^⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, docorigin, message⟩
96:             end if
97:             stop ⟨⟩, s′
98:         case else
99:             stop
100: end function
```

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. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

- The functions NAVBACK (Algorithm A.5) and NAVFORWARD (Algorithm A.6), navigate a window forward or backward. More precisely, they deactivate one document and activate that document's succeeding document or preceding document, respectively. If no such successor/predecessor exists, the functions do not change the state.

- The function RUNSCRIPT (Algorithm A.7) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). 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.

- The function PROCESSRESPONSE (Algorithm A.8) is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that

Algorithm A.8: Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, key, f, s')
2:     if Set-Cookie ∈ response.headers then
3:         for each c ∈^⟨⟩ response.headers[Set-Cookie], c ∈ Cookies do
4:             let s'.cookies[request.host]
                 ↪ := AddCookie(s'.cookies[request.host], c)
5:         end for
6:     end if
7:     if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
8:         let s'.sts := s'.sts +^⟨⟩ request.host
9:     end if
10:    if Referer ∈ request.headers then
11:        let referrer := request.headers[Referer]
12:    else
13:        let referrer := ⊥
14:    end if
15:    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
16:        let url := response.headers[Location]
17:        if url.fragment ≡ ⊥ then
18:            let url.fragment := requestUrl.fragment
19:        end if
20:        let method' := request.method
21:        let body' := request.body
22:        if Origin ∈ request.headers then
23:            let origin := ⟨request.headers[Origin], ⟨request.host, url.protocol⟩⟩
24:        else
25:            let origin := ⊥
26:        end if
27:        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
28:            let method' := GET
29:            let body' := ⟨⟩
30:        end if
31:        if ∃w̄ ∈ Subwindows(s') such that s'.w̄.nonce ≡ reference then    → Do not redirect XHRs.
32:            let req := ⟨HTTPReq, v6, method', url.host, url.path, url.parameters, ⟨⟩, body'⟩
33:            let referrerPolicy := response.headers[ReferrerPolicy]
34:            call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s')
35:        end if
36:    end if
```

This algorithm is continued on the next page.

```
37:     switch π1(reference) do
38:         case REQ
39:             let w̄ ← Subwindows(s′) such that s′.w̄.nonce ≡ π2(reference) if possible;
                ↪ otherwise stop    → normal response
40:             if response.body ≁ ⟨∗,∗⟩ then
41:                 stop {}, s′
42:             end if
43:             let script := π1(response.body)
44:             let scriptstate := π2(response.body)
45:             let d := ⟨ν7, requestUrl, response.headers, referrer, script, scriptstate, ⟨⟩, ⟨⟩, ⊤⟩
46:             if s′.w̄.documents ≡ ⟨⟩ then
47:                 let s′.w̄.documents := ⟨d⟩
48:             else
49:                 let ī ← ℕ such that s′.w̄.documents.ī.active ≡ ⊤
50:                 let s′.w̄.documents.ī.active := ⊥
51:                 remove s′.w̄.documents.(ī+1) and all following documents
                    ↪ from s′.w̄.documents
52:                 let s′.w̄.documents := s′.w̄.documents +⟨⟩ d
53:             end if
54:             stop {}, s′
55:         case XHR
56:             let w̄ ← Subwindows(s′), d̄ such that s′.d̄.nonce ≡ π2(reference)
                ↪ ∧ s′.d̄ = s′.w̄.activedocument if possible; otherwise stop
                  → process XHR response
57:             let headers := response.headers − Set-Cookie
58:             let s′.d̄.scriptinputs := s′.d̄.scriptinputs +⟨⟩
                ⟨XMLHTTPREQUEST, headers, response.body, π3(reference)⟩
59:             stop {}, s′
60: end function
```

was sent earlier. In *reference*, either a window or a document reference is given (see explanation for Algorithm A.4 above). *requestUrl* contains the URL used when retrieving the document.

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).

Browser Relation. We can now define the *relation $R_{webbrowser}$ of a Web browser atomic process* as follows:

Definition 58. The pair $((\langle\langle a, f, m\rangle\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm A.9 (or any of the functions called therein), when given $(\langle a, f, m\rangle, s)$ as input, termi-

Algorithm A.9: Web Browser Model: Main Algorithm.

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

let $s' := s$
if $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
  let $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests} \rangle$ → Collect incoming messages
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle \langle a', a, m' \rangle \rangle$, s'
end if
if $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
  let *switch* $\leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
  let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$
    ↪ **if possible; otherwise stop** → Pointer to some window.
  let $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$
    ↪ **if possible; otherwise stop** → Pointer to some top-level window.
  if *switch* $\equiv \texttt{script}$ **then** → Run some script.
    let $\overline{d} := \overline{w} +^{\langle\rangle} \texttt{activedocument}$
    call RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
  else if *switch* $\equiv \texttt{urlbar}$ **then** → Create some new request.
    let *newwindow* $\leftarrow \{\top, \bot\}$
    if *newwindow* $\equiv \top$ **then** → Create a new window.
      let *windownonce* $:= \nu_1$
      let $w' := \langle \textit{windownonce}, \langle\rangle, \bot \rangle$
      let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
    else → Use existing top-level window.
      let *windownonce* $:= s'.\overline{tlw}.\textit{nonce}$
    end if
    let *protocol* $\leftarrow \{\texttt{P}, \texttt{S}\}$
    let *host* $\leftarrow \mathsf{Doms}$
    let *path* $\leftarrow \mathbb{S}$
    let *fragment* $\leftarrow \mathbb{S}$
    let *parameters* $\leftarrow [\mathbb{S} \times \mathbb{S}]$
    let *url* $:= \langle \texttt{URL}, \textit{protocol}, \textit{host}, \textit{path}, \textit{parameters}, \textit{fragment} \rangle$
    let *req* $:= \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, \textit{host}, \textit{path}, \textit{parameters}, \langle\rangle, \langle\rangle \rangle$
    call HTTP_SEND(*windownonce*, *req*, *url*, $\bot$, $\bot$, $\bot$, s')
  else if *switch* $\equiv \texttt{reload}$ **then** → Reload some document.
    let *url* $:= s'.\overline{w}.\texttt{activedocument.location}$
    let *req* $:= \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, \textit{url}.\texttt{host}, \textit{url}.\texttt{path}, \textit{url}.\texttt{parameters}, \langle\rangle, \langle\rangle \rangle$
    let *referrer* $:= s'.\overline{w}.\texttt{activedocument.referrer}$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\texttt{nonce}, s')$
    call HTTP_SEND($s'.\overline{w}.\texttt{nonce}$, *req*, *url*, $\bot$, *referrer*, $\bot$, s')
  else if *switch* $\equiv \texttt{forward}$ **then**
    NAVFORWARD($\overline{w}$, s')
  else if *switch* $\equiv \texttt{back}$ **then**
    NAVBACK($\overline{w}$, s')
  end if

This algorithm is continued on the next page.

43: **else if** $m \equiv \texttt{FULLCORRUPT}$ **then** → Request to corrupt browser
44: **let** $s'.\texttt{isCorrupted} := \texttt{FULLCORRUPT}$
45: **stop** $\langle\rangle, s'$
46: **else if** $m \equiv \texttt{CLOSECORRUPT}$ **then** → Close the browser
47: **let** $s'.\texttt{secrets} := \langle\rangle$
48: **let** $s'.\texttt{windows} := \langle\rangle$
49: **let** $s'.\texttt{pendingDNS} := \langle\rangle$
50: **let** $s'.\texttt{pendingRequests} := \langle\rangle$
51: **let** $s'.\texttt{sessionStorage} := \langle\rangle$
52: **let** $s'.\texttt{cookies} \subset^{\langle\rangle} \text{Cookies}$ **such that**
↪ $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content.session} \equiv \bot)$
53: **let** $s'.\texttt{isCorrupted} := \texttt{CLOSECORRUPT}$
54: **stop** $\langle\rangle, s'$
55: **else if** $\exists \langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
↪ $\pi_1(\text{dec}_\text{s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
56: **let** $m' := \text{dec}_\text{s}(m, key)$
57: **if** $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
58: **stop**
59: **end if**
60: **remove** $\langle reference, request, url, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
61: **call** PROCESSRESPONSE(m', *reference*, *request*, *url*, *key*, f, s')
62: **else if** $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists \langle reference, request, url, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
↪ $m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
63: **remove** $\langle reference, request, url, \bot, f\rangle$ **from** $s'.\texttt{pendingRequests}$
64: **call** PROCESSRESPONSE(m, *reference*, *request*, *url*, *key*, f, s')
65: **else if** $m \in \text{DNSResponses}$ **then** → Successful DNS response
66: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \text{IPs} \vee m.\texttt{domain} \not\equiv \pi_2(s.\texttt{pendingDNS}).\texttt{host}$ **then**
67: **stop**
68: **end if**
69: **let** $\langle reference, message, url\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
70: **if** $url.\texttt{protocol} \equiv \text{S}$ **then**
71: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
↪ $+^{\langle\rangle} \langle reference, message, url, v_3, m.\texttt{result}\rangle$
72: **let** $message := \text{enc}_\text{a}(\langle message, v_3\rangle, s'.\texttt{keyMapping}[message.\texttt{host}])$
73: **else**
74: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
↪ $+^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
75: **end if**
76: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
77: **stop** $\langle\langle m.\texttt{result}, a, message\rangle\rangle, s'$
78: **end if**
79: **stop**

nates 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).

A.6.4. Definition of Web Browsers

Finally, we define Web browser atomic Dolev-Yao processes as follows:

Definition 59 (Web Browser atomic Dolev-Yao Process). A Web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, {s_o}^p)$ for a set I^p of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state ${s_0}^p \in Z_{\text{webbrowser}}$. ◇

A.6.5. Script Notations and Helper Functions

In order to simplify the description of scripts, we use several notations and helper functions. In the formal description of the scripts we use an abbreviation for URLs. We write URL^d_{path} to describe the following URL term: $\langle \mathtt{URL}, \mathtt{S}, d, path, \langle\rangle\rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write $\mathsf{URL}^{\mathsf{P}}_{path}$. For a (secure) origin $\langle d, \mathtt{S}\rangle$ of some domain d, we also write origin_d. Again, if the domain d belongs to some distinguished process P and d is the only domain associated to this process, we may write $\mathsf{origin}_{\mathsf{P}}$.

CHOOSEINPUT (Algorithm A.10). As explained in Section 2.5, the state of a document contains a term, say *scriptinputs*, which records the input this document has obtained so far (via XHRs and postMessages). If the script of the document is activated, it will typically need to pick one input message from *scriptinputs* and record which input it has already processed. For this purpose, the function $\mathsf{CHOOSEINPUT}(s', scriptinputs)$ is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate s' and chooses a yet unhandled input message from *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

CHOOSEFIRSTINPUTPAT (Algorithm A.11). Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *scriptinputs*, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in *scriptinputs* that matches *pattern* and returns it. This function is typically used in places, where a script only

Algorithm A.10: Function to retrieve an unhandled input message for a script.

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

Algorithm A.11: Function to extract the first script input message matching a specific pattern.

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

processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

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

PARENTDOCNONCE. The function PARENTDOCNONCE($tree, docnonce$) determines (similar to PARENTWINDOW above) the nonce referencing the active document in the parent window in the browser . 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, PARENTDOCNONCE 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 SUBWINDOWS($tree, docnonce$) returns $\langle\rangle$. Otherwise, let $\langle docnonce, origin, script, scriptstate, scriptinputs, subwindows, active\rangle$ denote the subterm of *tree* corresponding to the document referred to by *docnonce*. Then, SUBWINDOWS($tree, docnonce$) returns *subwindows*.

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

AUXDOCNONCE. Similar to AUXWINDOW above, the function AUXDOCNONCE takes a term *tree* and a document nonce *docnonce* as input. 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* or *docnonce* is not a top-level window, $\Diamond$ 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 $\mathsf{GETORIGIN}(\mathit{tree}, \mathit{docnonce})$ 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.

A.7. DNS Servers

Definition 60. 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

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

DNS queries are answered according to this table (if the requested DNS name cannot be found in the table, the request is ignored). $\diamond$

The relation $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of d above is defined by Algorithm A.12.

Algorithm A.12: Relation of a DNS server R^d.

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

A.8. Web Systems

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 61. A *Web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ 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 $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a Web attacker $p \in \mathsf{Web}$, 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 $p' \in \mathsf{Hon} \cup \mathsf{Web}$. 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 $p \in \mathsf{Hon}$ is a DY process which models either a *Web server*, a *Web browser*, or a *DNS server*. 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 $\mathsf{script}(s)$.

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

We now formally state and prove the general application independent security properties of the Web that have been sketched in Section 2.9.

Let $\mathcal{Web} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ be a Web system. In the following, we write $s_x = (S_x, E_x)$ for the states of a Web system.

Definition 62 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0,E^0,N^0),\ldots,(S^n,E^n,N^n))$ or an infinite run $\rho = ((S^0,E^0,N^0),\ldots)$ we say that *p emits e* iff there is a processing step in ρ of the form

$$(S^i,E^i,N^i) \xrightarrow[p\to E]{} (S^{i+1},E^{i+1},N^{i+1})$$

for some $i \geq 0$ and a set of events E with $e \in E$. We also say that *p emits m* iff $e = \langle x,y,m\rangle$ for some addresses x, y. ◇

Definition 63. We say that a term *t is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \to s_{i+1}$ of a run $\rho = (s_0,s_1,\ldots)$)* if t is derivable from t' with the knowledge available to P, i.e.,

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

◇

Definition 64. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. 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 65. 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 66. 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 A.8). ◇

Definition 67. 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., $t \in d_{\emptyset}(S(p))$. ◇

Definition 68. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm A.7) 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 $\rho = s_0, s_1, \ldots$ of any $\mathcal{Web}$, we state the following lemmas:

Lemma 1. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{Web}$ an honest browser b (I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(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 $\mathcal{Web}$ where any party except for u knows k', thus no one except for u can decrypt *req*.

(2) If there is a processing step $s_j \to s_{j+1}$ where the browser b leaks k to $\mathcal{W} \setminus \{u, b\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ or the browser is fully corrupted in s_j.

(3) The value of the host header in *req* is the domain that is assigned the public key $\mathsf{pub}(k')$ in the browsers' keymapping $s_0.\texttt{keymapping}$ (in its initial state).

(4) If b accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and b is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ 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 process b 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 b leaks k to $\mathcal{W} \setminus \{u,b\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and b and that the browser is not fully corrupted in s_j, and lead this to a contradiction.

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder v_3 in Lines 65ff. of Algorithm A.9) that is not used anywhere else. Further, the key is stored in the browser's state in *pendingRequests*. The information from *pendingRequests* is not extracted or used anywhere else (in particular it is not accessible by scripts). If the browser becomes closecorrupted prior to s_j (and after s_i), the key cannot be used anymore (compare Lines 46ff. of Algorithm A.9). Hence, b does not leak k to any other party in s_j (except for u and b). This proves (2).

(3) Per the definition of browsers (Algorithm A.9), a host header is always contained in HTTP requests by browsers. From Line 72 of Algorithm A.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 the browser's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by b as a response to m has to be encrypted with k. The nonce k is stored by the browser in the *pendingRequests* state information. The browser 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 b (which did not leak it either, as u did not leak it and b is honest, see (2)). The browser b cannot send responses. This proves (4). □

Corollary 1. In the situation of Lemma 1, as long as u does not leak the symmetric key k to $\mathcal{W} \setminus \{u,b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{b,u\}$ (i.e., $\nexists s' = (S',E') \in \rho$: $k \in d_{N^p}(S'(p))$).

Lemma 2. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \mathtt{S} \rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 45 in Algorithm A.8) the script in that document from an HTTPS response that was created by p.

Proof. The origin of the document d is set only once: In Line 45 of Algorithm A.8. The values (domain and protocol) used there stem from the information about the request (say, *req*) that led

to loading of d. These values have been stored in *pendingRequests* between the request and the response actions. The contents of *pendingRequests* are indexed by freshly chosen nonces and can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request *req* was added to *pendingRequests* in Line 71 (or Line 74 which we can exclude as we will see later) of Algorithm A.9. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in *pendingRequests*. When receiving the response to *req*, it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say, n). Only then the protocol part of the origin of the newly created document becomes `S`. The domain part of the origin (in our case *dom*) is taken directly from the *pendingRequests* and is thus guaranteed to be unaltered.

From Line 72 of Algorithm A.9 we can see that the encryption key for the request *req* was actually chosen using the host header of the message which will finally be the value of the origin of the document d. Since b therefore selects the public key $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \texttt{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that *req* was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 1 we see that the symmetric encryption key for the response, k, is only known to b and the respective Web server. The same holds for the nonce n that was chosen by the browser and included in the request. Thus, no other party than p can encrypt a response that is accepted by the browser b and which finally defines the script of the newly created document. □

Lemma 3. If in a processing step $s_i \rightarrow s_{i+1}$ of a run ρ of $\mathcal{Web}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \texttt{S} \rangle$ where and $S_i(b).\texttt{keyMapping}[dom] \equiv \texttt{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then that request was initiated by a script that b extracted (in Line 45 in Algorithm A.8) from an HTTPS response that was created by p.

Proof. First, we can see that the request was initiated by a script: As it contains an origin header, it must have been a POST request (see the browser definition in Appendix A.6). POST requests can only be initiated in Lines 55, 79 of Algorithm A.7 and Line 34 of Algorithm A.8. In the latter instance (Location header redirect), the request contains at least two different origins, therefore it is impossible to create a request with exactly the origin $\langle dom, \texttt{S} \rangle$ using a redirect. In the other two cases (FORM and XMLHTTPRequest), the request was initiated by a script.

The Origin header of the request is defined by the origin of the script's document. With Lemma 2 we see that the content of the document, in particular the script, was indeed provided by p. □

C. Auxiliary Definitions for Privacy

The definitions below complete our description of privacy of Web systems presented in Section 3.4.3.

Definition 69 (Web System Command and Schedule). We call a term ζ a *Web system command* (or simply, command) if ζ is of the form

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

The components are defined as follows:

- $i \in \mathbb{N}$,
- $j \in \mathbb{N}$,
- $cmd_{\text{switch}} \in \{1,2,3\}$,
- $cmd_{\text{window}} \in \mathbb{N}$,
- $\tau_{\text{script}} \in \mathcal{T}_{\emptyset}(V_{\text{script}} \cup \{x\})$ with x being a variable and V_{script} the set of placeholders for scripting processess (see Definition 46).
- $\tau_{\text{process}} \in \mathcal{T}_{\emptyset}(V_{\text{process}} \cup \{x\})$ with x being a variable and V_{process} the set of placeholders (see Definition 22).
- $url \in \mathsf{URLs}$ with URLs being the set of all valid URLs (see Definition 32).

We call a (finite) sequence $\sigma = \langle \zeta_1, \ldots, \zeta_n \rangle$, with $\zeta_1, \ldots, \zeta_n$ being Web system commands, a *Web system schedule* (or simply, schedule). ◇

Definition 70 (Induced Processing Step). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be a Web system and

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

be a processing step of $\mathcal{W}$ (as in Definition 41) with $E = (e_1, e_2, \ldots)$ and

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

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. $E' = E_{\text{out}} \cdot (e_1, \ldots, e_{i-1}, e_{i+1}, \ldots)$.
4. If p is a (Web) attacker process or p is a corrupted browser (i.e., $S(p).\texttt{isCorrupted} \not\equiv \bot$), then $E_{\text{out}} = \langle e_{\text{out}} \rangle$ with $\langle S'(p), e_{\text{out}} \rangle = \tau_{\text{process}}[\langle e_i, s \rangle / x]\downarrow$.
5. If p is an honest browser (i.e., $S(p).\texttt{isCorrupted} \equiv \bot$) and $m \equiv \texttt{TRIGGER}$, the browser relation behaves as follows and E_{out} and $S'(p)$ are obtained accordingly:
   a) If $cmd_{\text{switch}} = \texttt{script}$, the browser relation chooses $switch = \texttt{script}$ in Line 9 of Algorithm A.9 and $\overline{w}$ in Line 10 of Algorithm A.9 such that $\overline{w}$ is the cmd_{window}-th window in the tree of browser's state $S(p).\texttt{windows}$. If this script is not the attacker script, the browser (deterministically) executes the script in this window. Otherwise, in Line 10 of Algorithm A.7, the browser relation chooses the output of the script (of this window) as $out^{\lambda} = \tau_{\text{script}}[in/x]\downarrow$ with the variable in (deterministically) chosen in Line 9 of Algorithm A.7.
   b) If $cmd_{\text{switch}} = \texttt{urlbar}$, the browser relation chooses $switch = \texttt{urlbar}$ in Line 9 of Algorithm A.9 and $protocol$, $host$, $domain$, $path$, $parameters$ in Line 24ff. of Algorithm A.9 such that $url = \langle \texttt{URL}, protocol, host, path, parameters \rangle$.
   c) If $cmd_{\text{switch}} = \texttt{reload}$, the browser relation chooses $switch = \texttt{reload}$ in Line 9 of Algorithm A.9 and $\overline{w}$ in Line 10 of Algorithm A.9 such that $\overline{w}$ is the cmd_{window}-th window in the tree of browser's state $S(p).\texttt{windows}$. (The browser then starts to reload the document in this window.)
   d) If $cmd_{\text{switch}} = \texttt{forward}$, the browser relation chooses $switch = \texttt{forward}$ in Line 9 of Algorithm A.9 and $\overline{w}$ in Line 10 of Algorithm A.9 such that $\overline{w}$ is the cmd_{window}-th window in the tree of browser's state $S(p).\texttt{windows}$. The document history of that window is then changed such that the subsequent document is marked as the active document of that window. If there is no subsequent document in the window's history, nothing is changed.

e) If $cmd_{\text{switch}} = \texttt{back}$, the browser relation chooses $switch = \texttt{back}$ in Line 9 of Algorithm A.9 and $\overline{w}$ in Line 10 of Algorithm A.9 such that $\overline{w}$ is the cmd_{window}-th window in the tree of browser's state $S(p).\texttt{windows}$. The document history of that window is then changed such that the previous document is marked as the active document of that window. If there is no previous document in the window's history, nothing is changed.

We write

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

◇

Corollary 2. In some cases a command $\sigma = \langle i, j, \tau_{\text{process}}, cmd_{\text{switch}}, cmd_{\text{window}}, \tau_{\text{script}}, url \rangle$ 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, $cmd_{\text{switch}} \in \{1,3\}$, and $cmd_{\text{window}} > |\mathsf{Subwindows}(S(p))|$ (i.e., the command chooses a window of the browser p, which does not exist), then no processing step can be induced.

Definition 71 (Induced Run). Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be a Web system, $\sigma = \langle \zeta_1, \ldots, \zeta_n \rangle$ be a finite Web system schedule, and N^0 be an infinite sequence of pairwise disjoint nonces. We say that a finite run $\rho = ((S^0,E^0,N^0), \ldots, (S^n,E^n,N^n))$ 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 $\sigma(\mathcal{WS})$. ◇

Definition 72 (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. ◇

The definition of static equivalence follows the notion of static equivalence by Abadi and Fournet [AF01].

Definition 73 (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 $M, N \in \mathcal{T}_{\emptyset}(\{x\})$

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 74 (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 A.6) with two exceptions: (1) the state contains one more property, namely *challenge*, which initially contains the term $\top$. (2) Algorithm A.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., *challenge* $\not\equiv \bot$), then m is changed to be addressed to the domain dr and *challenge* is set to $\bot$ to recorded that a message was addressed to `CHALLENGE`. ◇

D. Model and Analysis of BrowserID in Primary Mode

D.1. Step-By-Step Description of BrowserID's Primary Mode

We now present additional details of the JavaScript implementation of BrowserID. While the basic steps have been shown in Section 4.2, we will now again refer to the steps denoted in Figure 4.3 and provide a step-by-step description. As above, for readability, we focus on the main login flow without the CIF, and we leave out steps for fetching additional resources (like JavaScript files) and some less relevant postMessages and XHRs. Figure D.1 depicts the same flow, but with even more details, such as the communication with the CIF.

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

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

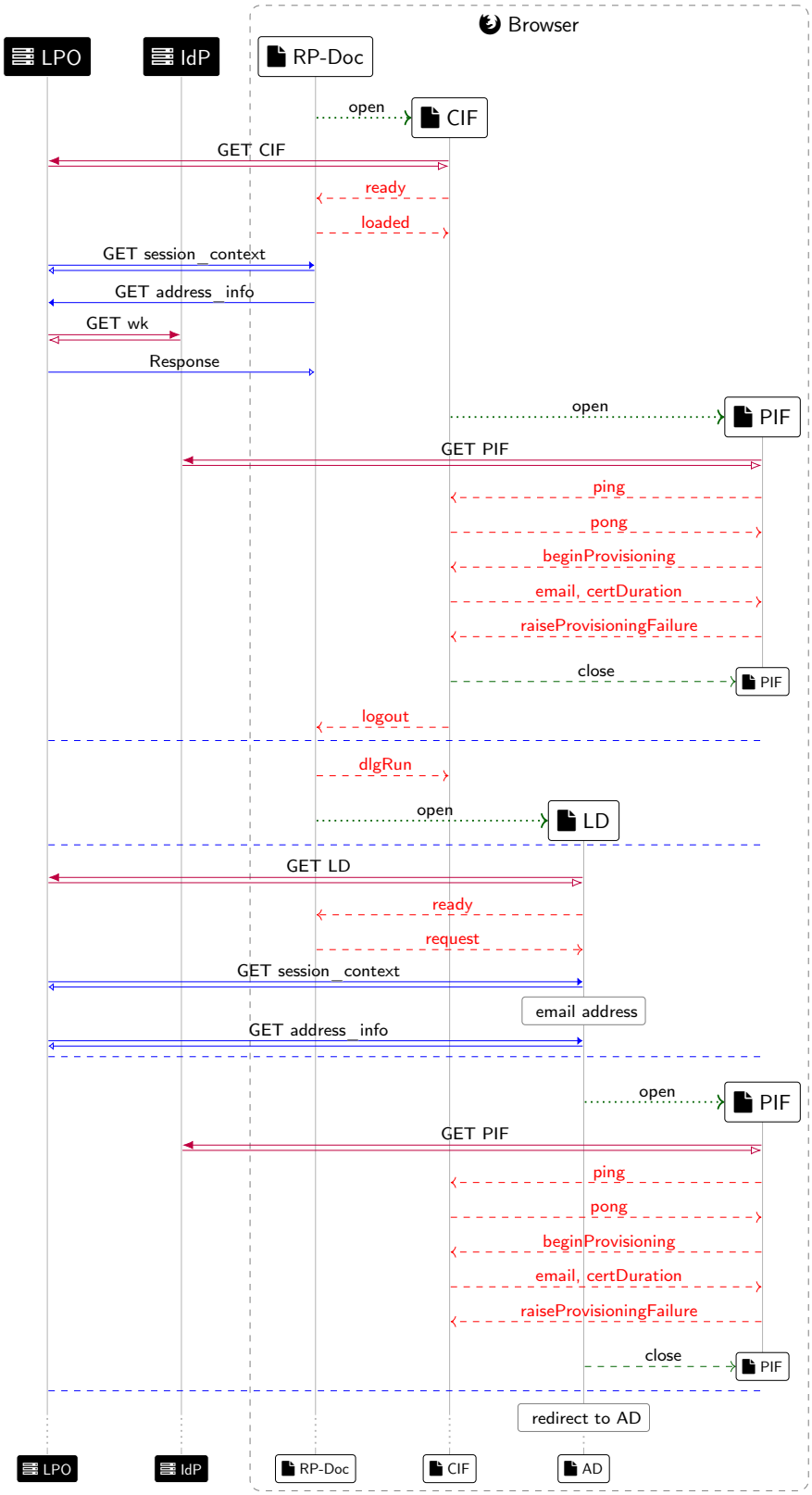


Figure D.1.: Detailed BrowserID implementation overview (primary mode). (See Page 11 for notation.)

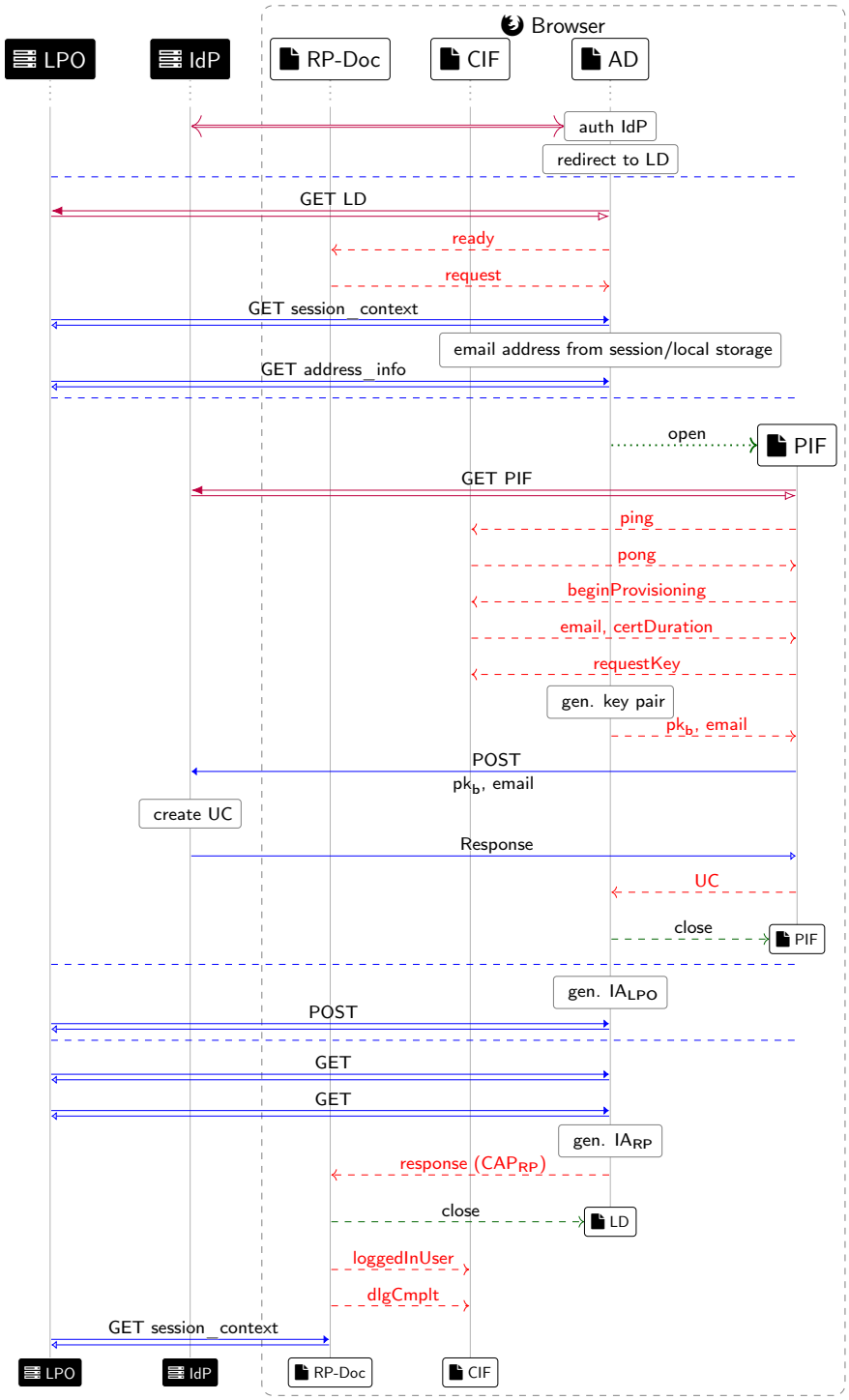


Figure D.1.: Detailed BrowserID implementation overview (primary mode, cont'd).

IdP. This document contains the public key of IdP, and two paths, the provisioning path and the authentication path at IdP. These paths will be used later in the login process by LD. LPO converts these paths into URLs and sends them in its response [9] to the requesting XHR [7].

Phase (ii). As there is no record about the login email address in the localStorage under the origin of LPO, the LD now tries to get a UC for this identity. For that to happen, the LD creates a new iframe, the *provisioning iframe* (PIF) [10]. The PIF's document is loaded [11] from the provisioning URL LD has just received before in [9]. The PIF now interacts with the LD via postMessages [12]. As the user is currently not logged in, the PIF tells the LD that the user is not authenticated yet. This also indicates to the LD that the PIF has finished operation. The LD then closes the PIF [13].

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

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

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

Phase (vi). The LD is now able to create a CAP, as it has access to a UC and the corresponding private key in its localStorage. First, LD creates an IA for LPO [21]. The IA and the UC is then combined to a CAP, which is then sent to LPO in an XHR POST message [22]. LPO is now able

to verify this CAP with the public key of IdP, which LPO has already fetched and cached before in 8. If the CAP is valid, LPO considers its session with the user's browser to be authenticated for the email address the UC in the CAP is issued for.

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

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

D.2. Sideshow/BigTent OpenID Flow

We will now give concrete examples of an OpenID flow that is started when Sideshow or BigTent want to authenticate a user via OpenID (as presented in Section 4.2). We will show typical requests and responses in such a flow and discuss the parameters that are important for the attacks presented in Section 4.5.1. We focus on Sideshow (and thus, Google), the URLs for BigTent (with Yahoo) are similar.

D.2.1. OpenID Authentication Request

As we already discussed in Section 4.2, Sideshow maintains a session with the user. Sideshow issues UCs to a user only if the session is authenticated. This authentication is done via OpenID. When detecting that a session is not authenticated, Sideshow redirects the user's browser to the so-called *OpenID endpoint URL* of Google/Gmail. This URL may look as follows (line breaks added for readability):

```
https://www.google.com/accounts/o8/ud
 ?openid.mode=checkid_setup
 &openid.ns=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0
 &openid.ns.ax=http%3A%2F%2Fopenid.net%2Fsrv%2Fax%2F1.0
 &openid.ax.mode=fetch_request
 &openid.ax.type.email=http%3A%2F%2Faxschema.org%2Fcontact%2Femail
 &openid.ax.required=email
```

```
&openid.ns.ui=http%3A%2F%2Fspecs.openid.net%2Fextensions%2Fui%2F1.0
&openid.ui.mode=popup
&openid.identity=
 http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select
&openid.claimed_id=
 http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select
&openid.return_to=
 https%3A%2F%2Fgmail.login.persona.org%2Fauthenticate%2Fverify
&openid.realm=https%3A%2F%2F*.persona.org
```

In this URL, the parameter `openid.ax.type.email` encodes that Sideshow requests under the name "email" in the namespace "ax" an attribute of the type `http://axschema.org/contact/email`. Per definition of the OpenID attribute exchange schema [HBH07], this denotes the request for an email address. Note that Google's OpenID endpoint is (per the OpenID protocol) not obliged to follow this request and may issue an OpenID assertion without a (signed) email address.

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

D.2.2. OpenID Authentication Response

After accessing the above OpenID endpoint URL, the user authenticates with Google and confirms that Google releases the requested information to Sideshow. Google then creates an OpenID assertion and appends it to the `openid.return_to` URL that was contained in the OpenID authentication request. Finally, OpenID redirects the user's browser to the resulting URL, which may look as follows:

```
https://gmail.login.persona.org/authenticate/verify
 ?openid.ns=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0
 &openid.mode=id_res
 &openid.op_endpoint=https%3A%2F%2Fwww.google.com%2Faccounts%2Fo8%2Fud
 &openid.response_nonce=2013-09-24T11%3A46%3A11Z479iYHqAdS054A
 &openid.return_to=https%3A%2F%2Fgmail.login.persona.org%2Fauthenticate%2Fverify
 &openid.assoc_handle=1.AMlYA(...)ubxCOqB
 &openid.signed=op_endpoint
  %2Cclaimed_id
  %2Cidentity
  %2Creturn_to
  %2Cresponse_nonce
  %2Cassoc_handle
```

```
 %2Cns.ext1
 %2Cext1.mode
 %2Cext1.type.email
 %2Cext1.value.email
&openid.sig=BIPe1PIwitMp365MUEtd34IJLUs%3D
&openid.identity=
 https%3A%2F%2Fwww.google.com%2Faccounts%2Fo8%2Fid%3Fid%3DAItO(...)sLfSiQ
&openid.claimed_id=
 https%3A%2F%2Fwww.google.com%2Faccounts%2Fo8%2Fid%3Fid%3DAItO(...)sLfSiQ
&openid.ns.ext1=http%3A%2F%2Fopenid.net%2Fsrv%2Fax%2F1.0
&openid.ext1.mode=fetch_response
&openid.ext1.type.email=http%3A%2F%2Faxschema.org%2Fcontact%2Femail
&openid.ext1.value.email=user%40gmail.com
&openid.ns.ext2=http%3A%2F%2Fspecs.openid.net%2Fextensions%2Fui%2F1.0
&openid.ext2.mode=popup
```

First note that the namespace "ax" was renamed by Gmail: What was prefixed with `openid.ax` in the request is now prefixed with `openid.ext1`. The parameter `openid.signed` contains the names of the parameters that have actually been MACed into the signature given in `openid.sig`.

Note that the receiver of the assertion can not know the exact renaming performed by Gmail and must, although the renaming is obvious in this case, rely on the type parameters to determine the actual contents of the parameters. In this case, `openid.ext1.type.email` contains the AX schema type for an email address, saying that `openid.ext1.value.email` actually contains the requested email address.

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

D.2.3. Verification

After receiving the above request, Sideshow can forward all attributes to a verification service at Google. The URL for this verification is `https://www.google.com/accounts/o8/ud?openid.mode=checkid_authentication` (same as in the authentication request, but with a different `openid.mode` value). Sideshow appends to this URL all attributes from the authentication response, i.e., the assertion, except for the `openid.mode` parameter. Google (only) checks that the MAC in `openid.sig` is correct (using the symmetric key stored earlier) and answers with "true" or "false" accordingly.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_rp_index	`script_rp_index`
script_lpo_cif	`script_lpo_cif`
script_lpo_ld	`script_lpo_ld`
script_idp_pif	`script_idp_pif`
script_idp_ad	`script_idp_ad`

Table D.1.: List of scripts in $\mathcal{S}$ and their respective string representations for the primary mode of BrowserID.

D.3. Model of BrowserID in Primary Mode

We now present full details of our formal model of BrowserID in primary mode that we outlined in Section 4.4.1. As mentioned, this model incorporates the fixes discussed in Section 4.5.1. We model the BrowserID system as an SSO Web system (in the sense of Section 3.1). We call a Web system $\mathcal{BID}^p = (\mathcal{W}^p, \mathcal{S}, \mathsf{script}, E_0)$ a *BrowserID primary mode Web system* if it is of the form outlined in Section 4.4.1 and described in what follows.

Recall that the system $\mathcal{W}^p = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$ contains the Web server for LPO, a finite set B of Web browsers, a finite set RP of Web servers for the relying parties, and a finite set IDP of Web servers for the identity providers, with

$$\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP} \cup \mathsf{DNS} \cup \underbrace{\{\mathsf{LPO}\}}_{=:\mathsf{Other}} .$$

For the analysis of authentication and session integrity, we consider a BrowserID primary mode Web system with one network attacker, i.e., $\mathsf{Web} := \emptyset$, and $\mathsf{Net} := \{\mathsf{attacker}\}$. In this case, DNS servers are assumed to be dishonest, and hence, are subsumed by attacker. (For the analysis of privacy, we consider a BrowserID primary mode Web system for privacy analysis, that follows the description in Section 3.4.3. As privacy in BrowserID is broken beyond repair, we here omit a formal definition as we cannot formally prove this property.) More details on the processes in $\mathcal{W}^p$ are provided below. Table D.1 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 A.8.

D.3.1. Addresses and Domain Names

The set IPs contains for LPO, attacker, every relying party in RP, every identity provider in IDP, and every browser in B one address each. By addr we denote the corresponding assignment

from a process to its address. The set Doms contains one domain for LPO, one for every relying party in RP, a finite set of domains for every identity provider in IDP, and a finite set of domains for attacker. Browsers (in B) do not have a domain. By dom we denote the assignments from atomic processes to sets of Doms. If dom or addr returns a set with only one element, we often write $\mathsf{dom}(x)$ or $\mathsf{addr}(x)$ to refer to the element.

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

D.3.2. Keys and Secrets

The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey} \colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. Atomic processes are given the private keys for their domain in the following form: For an atomic DY process p we define

$$tlskeys^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle \ .$$

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

The set $\mathsf{Secrets} \subseteq \mathcal{N}$ is the set of passwords (secrets) the browsers share with the identity providers.

We note that while initially only browsers own secrets and identities, the attacker can corrupt any number of browsers in a run, and hence, can own secrets and identities as well.

D.3.3. Corruption

RPs and IdPs 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 or IdP is *honest* if the according part of their state ($s.\texttt{corrupt}$) is $\bot$, and that they are corrupted otherwise.

Recall that browsers can, as explained in Section 2.5, become corrupted as well.

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

D.3.4. Attackers

As mentioned, the attacker attacker is modeled to be a network or Web attacker (depending on whether it is a system with a network attacker or a system for privacy analysis) as specified in Section 2.4. We allow network attackers to listen to/spoof all available IP addresses, and hence,

define $I^{\mathsf{attacker}} = \mathsf{IPs}$. Web attackers get an IP address disjoint from other IP addresses just like any honest process. His initial state is $s_0^{\mathsf{attacker}} = \langle attdoms, tlskeys, signkeys \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker, *tlskeys* is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all verification keys for all IdPs. Note that the attacker can also corrupt IdPs in order to obtain valid (private) signing keys.

In a system with a network attacker, all other parties use the attacker as a DNS server.

D.3.5. Browsers

Each $b \in \mathsf{B}$ is a Web browser as defined in Appendix A.6, with $I^b := \{\mathsf{addr}(b)\}$ being its address.

To define the inital state, first let $ID^b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all identities of b, $ID^{b,d} := \{i \mid \exists x:\ i = \langle x, d \rangle \in ID^b\}$ be the set of identities of b for a domain d, and $SecretDomains^b := \{d \mid ID^{b,d} \neq \emptyset\}$ 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 (TLS) key, according to the mapping tlskey; the DNS address is $\mathsf{addr}(\mathtt{attacker})$; the list of secrets contains an entry $\langle \langle d, \mathtt{S} \rangle, s \rangle$ for each $d \in SecretDomains^b$ and $s = \mathsf{secretOfID}(i)$ for some $i \in ID^{b,d}$ (s is the same for all i); *ids* is $\langle ID^b \rangle$; *sts* is empty.

D.3.6. LPO

LPO is modeled as an atomic DY process $(I^{\mathsf{LPO}}, Z^{\mathsf{LPO}}, R^{\mathsf{LPO}}, s_0^{\mathsf{LPO}})$ with the IP address $I^{\mathsf{LPO}} = \{\mathsf{addr}(\mathsf{LPO})\}$. The initial state s_0^{LPO} of LPO contains the private key of its domain, and the signing keys of all IdPs (LPO does not need the public tls keys of other parties, which is why we omit them from LPO's initial state.). The definition of R^{LPO} follows the description of LPO in Section 4.2 and is provided below.

HTTP responses by LPO can contain strings that represent scripts (`script_lpo_cif` and `script_lpo_ld`). These scripts are defined in Appendix D.3.9 below.

In order to define the set Z^{LPO} of states of LPO, we first define the terms describing the session context of a session.

Definition 75. A term of the form $\langle ids, xsrfToken \rangle$ with $ids \subset^{\langle\rangle} \mathsf{ID}$ and $xsrfToken \in \mathcal{N}$ is called an *LPO session context*. We denote the set of all LPO session contexts by LPOSessionCTXs. $\diamond$

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

Definition 76. A *state* $s \in Z^{\mathsf{LPO}}$ *of LPO* is a term of the form $\langle tlskey, signkeys, sessions\rangle$ where $tlskey = \mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO}))$, *signkeys* is a mapping of domains to signing keys of the form

$$signkeys = \langle\{\langle d, \mathsf{pub}(\mathsf{signkey}(y))\rangle \mid y \in \mathsf{IdPs},\ d \in \mathsf{dom}(y)\}\rangle\ ,$$

and $sessions \in [\mathcal{N} \times \mathsf{LPOSessionCTXs}]$.[1]

The *initial state* s_0^{LPO} *of LPO* is a state of LPO with $s_0^{\mathsf{LPO}}.\mathtt{sessions} = \langle\rangle$. ◇

Example 8. Let k be a signing key for some IdP which owns the domain `example.com`. A possible state s of LPO may look like this:

$$s = \langle\langle n_1, \ldots, n_m\rangle, \mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO})), [\mathtt{example.com} : \mathsf{pub}(k)], sessions\rangle$$

with

$$sessions = \langle\langle \mathtt{sessionid}_1, \langle\langle id'_1, \ldots, id'_l\rangle, \mathtt{xsrfToken}\rangle\rangle, \ldots\rangle$$

The relation $R^{\mathsf{LPO}} \subseteq (\mathcal{E} \times Z^{\mathsf{LPO}}) \times (2^{\mathcal{E}} \times Z^{\mathsf{LPO}})$ of LPO is specified in Algorithm D.1. Just like in Appendix A.6, we describe this relation by a non-deterministic algorithm.

D.3.7. Identity Providers

An identity provider $i \in \mathsf{IDPs}$ is a Web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the address $I^i := \{\mathsf{addr}(i)\}$. Its initial state s_0^i contains a list of domains and (private) TLS keys (see below), a list of users and identites (see below), 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.

Sessions are structured as a dictionary: For each session identifier (session id) the dictionary contains the list of identities for which the session is authenticated.

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 "user database" $userset^i$ (for the identity provider i), we first define the set

$$\mathsf{Secrets}^i = \bigcup_{j \in \mathsf{ID}^i} \mathsf{secretOfID}(j)\ ,$$

the function

$$\mathsf{IDsofSecret} : \mathsf{Secrets} \to \mathsf{ID},\ s \mapsto \{j \mid j \in \mathsf{ID}\ ,\ \mathsf{secretOfID}(j) = s\}\ ,$$

[1] As mentioned before, the state of LPO does not need to contain public keys.

Algorithm D.1: Relation of LPO R^{LPO}.

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

1: **let** $s' := s$
2: **let** $sts := \langle \texttt{Strict-Transport-Security}, \top\rangle$
3: **if** $m \equiv \texttt{TRIGGER}$ **then**
4: **if** $s'.\texttt{sessions} \equiv \langle\rangle$ **then**
5: **stop**
6: **end if**
7: **let** $sessionid \leftarrow \{id \mid id \in^{\langle\rangle} s'.\texttt{sessions}\}$
8: **let** $choice \leftarrow \{\texttt{logout}, \texttt{expire}\}$
9: **if** $choice \equiv \texttt{logout}$ **then**
10: **let** $s'.\texttt{sessions}[sessionid].\texttt{ids} := \langle\rangle$
11: **else**
12: **let** $s'.\texttt{sessions} := s'.\texttt{sessions} - sessionid$
13: **end if**
14: **stop**
15: **end if**
16: **let** m_{dec}, k **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, s.\texttt{tlskey})$
↪ **if possible; otherwise stop**
17: **let** n, $method$, $path$, $params$, $headers$, $body$ **such that**
↪ $\langle \texttt{HTTPReq}, n, method, \mathsf{dom}(\mathsf{LPO}), path, params, headers, body\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
18: **if** $method \equiv \texttt{GET} \wedge path \equiv /\texttt{cif}$ **then** → Deliver CIF script
19: **let** $scriptinit := \langle \texttt{init}, \bot, \bot, \bot, \bot, \bot, \bot, \langle\rangle, \bot, \bot\rangle$
20: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts\rangle, \langle \texttt{script\_lpo\_cif}, scriptinit\rangle\rangle, k,)$
21: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
22: **else if** $method \equiv \texttt{GET} \wedge path \equiv /\texttt{ld}$ **then** → Deliver LD script
23: **let** $scriptinit := \langle \texttt{init}, \bot, \bot, \bot, \bot, \bot, \langle\rangle, \bot, \bot, \bot\rangle$
24: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts\rangle, \langle \texttt{script\_lpo\_ld}, scriptinit\rangle\rangle, k,)$
25: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
26: **else if** $method \equiv \texttt{GET} \wedge path \equiv /\texttt{ctx}$ **then** → Deliver context info.
27: **let** $sessionid := headers[\texttt{Cookie}][\texttt{browserid\_state}]$
28: **if** $sessionid \notin^{\langle\rangle} s.\texttt{sessions}$ **then**
29: **let** $sessionid := v_1$
30: **let** $xsrfToken := v_2$
31: **let** $s'.\texttt{sessions} := s'.\texttt{sessions} +^{\langle\rangle} \langle sessionid, \langle\langle\rangle, xsrfToken\rangle\rangle$
32: **end if**
33: **let** $context := \langle \bot, s'.\texttt{sessions}[sessionid].\texttt{xsrfToken}\rangle$
34: **if** $s'.\texttt{session}[sessionid].\texttt{ids} \not\equiv \langle\rangle$ **then**
35: **let** $context.1 := \top$
36: **end if**
37: **let** $setCookie := \langle \texttt{Set-Cookie}, \langle\langle \texttt{browserid\_state}, sessionid, \top, \top, \top\rangle\rangle\rangle$
38: **let** $headers := \langle sts, setCookie\rangle$
39: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, headers, context\rangle, k)$
40: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$

This algorithm is continued on the next page.

41: **else if** $method \equiv \texttt{POST} \wedge path \equiv /\texttt{auth}$ **then**
42: **let** uc, ia, $xsrfToken$ **such that** $\langle\langle uc, ia\rangle, xsrfToken\rangle \equiv body$ **if possible; otherwise stop**
43: **let** $sessionid := headers[\texttt{Cookie}][\texttt{browserid\_state}]$
44: **if** $s'.\texttt{sessions}[sessionid].\texttt{xsrfToken} \not\equiv xsrfToken$ **then**
45: **stop**
46: **end if**
47: **let** $name$, $domain$, $userpubkey$ **such that** $\langle\langle name, domain\rangle, userpubkey\rangle \equiv \mathsf{extractmsg}(uc)$ **if possible; otherwise stop**
48: **let** $id := \langle name, domain\rangle$
49: **let** $origin := \mathsf{extractmsg}(ia)$
50: **if** $\mathsf{checksig}(uc, s.\texttt{signkeys}[domain]) \not\equiv \top \vee \mathsf{checksig}(ia, userpubkey) \not\equiv \top \vee origin \not\equiv \langle s.\texttt{domain}, \mathsf{S}\rangle$ **then**
51: **stop**
52: **end if**
53: **if** $s'.\texttt{sessions}[sessionid].\texttt{ids} \equiv \langle\rangle$ **then**
54: **if** $\nexists n \in \mathbb{N}$ **such that** $id \in^{\langle\rangle} s'.\texttt{idgroups}.n$ **then**
55: **let** $s'.\texttt{idgroups} := s'.\texttt{idgroups} +^{\langle\rangle} \langle id\rangle$
56: **end if**
57: **let** $n \leftarrow \mathbb{N}$ **such that** $id \in^{\langle\rangle} s'.\texttt{idgroups}.n$
58: **else**
59: **let** $n \leftarrow \mathbb{N}$ **such that** $s'.\texttt{idgroups}.n \equiv s'.\texttt{sessions}[sessionid].\texttt{ids}$ **if possible; otherwise stop**
60: **if** $id \notin^{\langle\rangle} s'.\texttt{idgroups}.n$ **then**
61: **let** $s'.\texttt{idgroups}.n := s'.\texttt{idgroups}.n +^{\langle\rangle} \langle name, domain\rangle$
62: **end if**
63: **end if**
64: **let** $s'.\texttt{sessions}[sessionid].\texttt{ids} := s'.\texttt{idgroups}.n$
65: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts\rangle, \top\rangle, k)$
66: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
67: **end if**
68: **stop**

and finally

$$userset^i = \langle\{\langle s, \langle \mathsf{IDsofSecret}(s)\rangle\rangle \mid s \in \mathsf{Secrets}^i\}\rangle\ .$$

We also need a term that represents a dictionary that maps domains to (private) TLS keys of the IdP i. We define $tlskeys^i = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(i)\}\rangle$.

Definition 77. A *state $s \in Z^i$ of an IdP i* is a term of the form $\langle tlskeys, users, signkey, sessions, corrupt\rangle$ where $tlskeys = tlskeys^i$, $users = userset^i$, $signkey \in \mathcal{N}$ (the key used by the IdP i to sign UCs), $sessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$.

The *initial state s_0^i of i* is the state $\langle\langle\rangle, tlskeys^i, userset^i, \mathsf{signkey}(i), \langle\rangle, \bot\rangle$. ◇

The relation R^i that defines the behavior of the IdP i is defined by Algorithm D.2.

D.3.8. Relying Parties

A relying party $r \in \mathsf{RP}$ is a Web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the address $I^r := \{\mathsf{addr}(r)\}$. Its initial state s_0^r contains its domain, the private key associated with its domain, the DNS server address, and the verification keys of all IdPs.[2] The full state additionally contains the set of service tokens the RP has issued. The definition of R^r again follows the description in Section 4.2. RP accepts only HTTPS requests.

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 $I^r = \{\mathsf{addr}(r)\}$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 78. A *state $s \in Z^r$ of an RP r* is a term of the form

$$\langle domain, tlskey, signkeys, serviceTokens, corrupt\rangle$$

where $domain = \mathsf{dom}(r)$, $tlskey = \mathsf{tlskey}(\mathsf{dom}(r))$ the private TLS key for the domain of r, $signkeys = \langle\{\langle d, \mathsf{pub}(\mathsf{signkey}(y))\rangle \mid y \in \mathsf{IdPs},\ d \in \mathsf{dom}(y)\}\rangle$ (same as for LPO), $serviceTokens \in [\mathcal{N} \times \mathbb{S}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$.

The *initial state s_0^r of r* is a state of r with $s_0^r.\mathtt{serviceTokens} = \langle\rangle$ and $s_0^r.\mathtt{corrupt} = \bot$. ◇

As mentioned earlier, we define the function serviceSessions that extracts service sessions from an RP's state as the projection that extracts the subterm *serviceTokens*.

We specify the relation $R^r \subseteq (\mathcal{E} \times Z^r) \times (2^{\mathcal{E}} \times Z^r)$ of r in Algorithm D.3.

[2] We add the IdP verification keys to the initial status (instead of having RPs retrieve them dynamically from the IdP) in order to reduce the overall complexity.

Algorithm D.2: Relation of an IdP R^i.

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

1: **let** $s' := s$
2: **if** $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
3: **let** $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
4: **let** $m' \leftarrow d_{N^p}(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle\langle a', a, m'\rangle\rangle$, s'
7: **end if**
8: **let** $sts := \langle\texttt{Strict-Transport-Security}, \top\rangle$
9: **let** m_{dec}, k, k', *inDomain* **such that**
↪ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s.\texttt{tlskeys}$
↪ **if possible; otherwise stop**
10: **let** n, *method*, *path*, *params*, *headers*, *body* **such that**
↪ $\langle\texttt{HTTPReq}, n, method, inDomain, path, params, headers, body\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
11: **if** $method \equiv \texttt{POST}$ **then**
12: **if** $path \not\equiv /\texttt{certreq}$ **then** → User logs in.
13: **let** *id*, *secret* **such that** $\langle id, secret\rangle \equiv body$
↪ **if possible; otherwise stop**
14: **if** $headers \not\equiv \langle\texttt{Origin}, \langle inDomain, \texttt{S}\rangle\rangle$ **then**
15: **stop**
16: **end if**
17: **let** $ids := s.\texttt{users}[secret]$
18: **if** $ids \equiv \langle\rangle \vee id \equiv \langle\rangle \vee id \notin^{\langle\rangle} ids$ **then** → Check id/secret pair.
19: **stop**
20: **end if**
21: **let** $sessionid := \nu_1$
22: **let** $s'.\texttt{sessions}[sessionid] := ids$
23: **let** $setCookie := \langle\texttt{Set-Cookie}, \langle\langle\texttt{sessionid}, sessionid, \top, \top, \top\rangle\rangle\rangle$
24: **let** $m' := \mathsf{enc_s}(\langle\texttt{HTTPResp}, n, 200, \langle sts, setCookie\rangle, \top\rangle, k)$
25: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
26: **else** → User wants a certificate.
27: **let** *id*, *pubkey* **such that** $\langle id, pubkey\rangle \equiv body$
↪ **if possible; otherwise stop**
28: **let** $sessionid := headers[\texttt{Cookie}][\texttt{sessionid}]$
29: **if** $id \notin^{\langle\rangle} s'.\texttt{sessions}[sessionid]$ **then** → Check if user is logged in.
30: **stop**
31: **end if**
32: **let** $uc := \mathsf{sig}(\langle id, pubkey\rangle, s.\texttt{signkey})$
33: **let** $m' := \mathsf{enc_s}(\langle\texttt{HTTPResp}, n, 200, \langle sts\rangle, uc\rangle, k)$
34: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
35: **end if**
36: **else**

This algorithm is continued on the next page.

37: **if** $path \equiv /\texttt{pif}$ **then**
38: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts \rangle, \langle \texttt{script\_idp\_pif}, \hookrightarrow \langle \texttt{init}, \langle\rangle, \langle\rangle, \langle\rangle, \bot, \bot, \bot \rangle \rangle \rangle, k)$
39: **else**
40: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts \rangle, \langle \texttt{script\_idp\_ad}, \langle\rangle \rangle \rangle, k)$
41: **end if**
42: **stop** $\langle \langle f, a, m' \rangle \rangle$, s'
43: **end if**
44: **stop**

Algorithm D.3: Relation of an RP R^r.

Input: $\langle a, f, m \rangle, s$
1: **let** $s' := s$
2: **if** $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
3: **let** $s'.\texttt{corrupt} := \langle \langle a, f, m \rangle, s'.\texttt{corrupt} \rangle$
4: **let** $m' \leftarrow d_{NP}(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle \langle a', a, m' \rangle \rangle$, s'
7: **end if**
8: **let** $sts := \langle \texttt{Strict-Transport-Security}, \top \rangle$
9: **let** m_{dec}, k **such that** $\langle m_{\text{dec}}, k \rangle \equiv \mathsf{dec_a}(m, s.\texttt{tlskey})$ $\hookrightarrow$ **if possible; otherwise stop**
10: **let** n, *method*, *path*, *params*, *headers*, *body* **such that** $\hookrightarrow \langle \texttt{HTTPReq}, n, method, s.\texttt{domain}, path, params, headers, body \rangle \equiv m_{\text{dec}}$ $\hookrightarrow$ **if possible; otherwise stop**
11: **if** $method \equiv \texttt{GET}$ **then**
12: **let** $scriptinit := \langle \texttt{init}, \bot, \bot, \bot, \langle\rangle, \langle\rangle, \bot \rangle$
13: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts \rangle, \langle \texttt{script\_rp\_index}, scriptinit \rangle \rangle, k)$
14: **stop** $\langle \langle f, a, m' \rangle \rangle$, s'
15: **else if** $method \equiv \texttt{POST} \wedge headers \equiv \langle \texttt{Origin}, \langle s.\texttt{domain}, \texttt{S} \rangle \rangle$ **then**
16: **let** uc, ia **such that** $\langle uc, ia \rangle \equiv body$ **if possible; otherwise stop**
17: **let** *name*, *domain*, *userpubkey* **such that** $\hookrightarrow \langle \langle name, domain \rangle, userpubkey \rangle \equiv \mathsf{extractmsg}(uc)$ $\hookrightarrow$ **if possible; otherwise stop**
18: **let** $id := \langle name, domain \rangle$
19: **let** $origin := \mathsf{extractmsg}(ia)$
20: **if** $\mathsf{checksig}(uc, s.\texttt{signkeys}[domain]) \not\equiv \top \vee \mathsf{checksig}(ia, userpubkey) \not\equiv \top \vee$ $\hookrightarrow origin \not\equiv \langle s.\texttt{domain}, \texttt{S} \rangle$ **then**
21: **stop**
22: **end if**
23: **let** $n_{\text{token}} := \nu_1$
24: **let** $s'.\texttt{serviceTokens} := s'.\texttt{serviceTokens} +^{\langle\rangle} \langle n_{\text{token}}, id \rangle$
25: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle sts \rangle, \langle n_{\text{token}}, id \rangle \rangle, k)$
26: **stop** $\langle \langle f, a, m' \rangle \rangle$, s'
27: **end if**

D.3.9. BrowserID Scripts

As already mentioned above, the set $\mathcal{S}$ of the Web system $\mathcal{BID}^p_{\text{primary}} = (\mathcal{W}^p, \mathcal{S}, \mathsf{script}, E_0)$ consists of the scripts R^{att}, *script_rp_index*, *script_lpo_cif*, *script_lpo_ld*, *script_idp_pif*, and *script_idp_ad* with their string representations being defined by script (see Table D.1).

The script R^{att} is the attacker script (see Section 2.5). The formal model of the other scripts follows the description in Appendix D.1.

In what follows, we provide a definition for all honest scripts.

login.persona.org Communication Iframe Script (script_lpo_cif). Recall that a script is a relation that takes as input a term and outputs a new term. As described in Section 2.5 and formally specified in Algorithm A.7 ($m = \texttt{TRIGGER}$, $action = 1$), the input term is provided by the browser. It contains the scriptstate and additional information containing all browser state information the script has access to, such as the input the script has obtained so far via XHRs and postMessages, information about windows, etc. The browser expects the output term to have a specific form, as also specified in Section 2.5 and Algorithm A.7. The output term contains, among other information, the new internal scriptstate.

As for *script_lpo_cif*, this script models the script run in the CIF, as sketched in Appendix D.1.

We first describe the structure of the internal scriptstate of the script *script_lpo_cif*.

Definition 79. A *scriptstate s of script_lpo_cif* is a term of the form

$$\langle q, \mathit{requestOrigin}, \mathit{loggedInUser}, \mathit{pause}, \mathit{context}, \mathit{key}, \mathit{uc}, \mathit{handledInputs}, \mathit{refXHRctx}, \mathit{PIFindex} \rangle$$

where $q \in \mathbb{S}$, $\mathit{requestOrigin} \in \mathsf{Origins} \cup \{\bot\}$, $\mathit{loggedInUser} \in \mathsf{ID} \cup \{\langle\rangle, \bot\}$, $\mathit{pause} \in \{\top, \bot\}$, $\mathit{context} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{key} \in \mathcal{N} \cup \{\bot\}$, $\mathit{uc} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{handledInputs} \subset^{\langle\rangle} \mathbb{N}$, $\mathit{refXHRctx} \in \mathcal{N} \cup \{\bot\}$, $\mathit{PIFindex} \in \mathbb{N} \cup \{\bot\}$. The *initial scriptstate* $\mathit{initState}_{cif}$ of *script_lpo_cif* is the state

$$\langle \texttt{init}, \bot, \bot, \bot, \bot, \bot, \bot, \langle\rangle, \bot, \bot \rangle \,.$$

◇

Before we provide the formal specification of the relation of *script_lpo_cif*, we present an informal description that completes the description of scripts provided in Section 4.4.1.

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

$q = \texttt{default}$ This is the state to which `script_lpo_cif` always returns to. This state handles all postMessages the CIF expects to receive from its parent window. If the postMessage

received was sent from the parent window of the CIF, it behaves as follows, depending on the first element of the received postMessage:

postMessage `loaded` The script records the sender's origin of the received postMessage as the remote origin in the scriptstate if the scriptstate did not contain any information about the remote origin yet. Also, an identity, which represents the assumption of the sender on who it believes to be logged in, is saved in the scriptstate. If the *pause* flag in the scriptstate is $\top$ it transitions to the state `default`. Otherwise, it is checked, if the current context in the scriptstate is $\bot$. If the check is true, the script transitions to the state `fetchContext`, or to the state `checkAndEmit` otherwise.

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

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

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

postMessage `logout` The script removes the entry for the RP (recorded in the scriptstate) from the localStorage and then transitions to the state `sendLogout`. If no remote origin is set in the script's state, it is now set to the sender's origin of the received postMessage.

$q =$ `fetchContext` In this state, the script sends an XHR to LPO with a `GET` request to the path `/ctx` and then transitions to the state `receiveContext`.

$q =$ `receiveContext` In this state, the script expects an XHR response as input containing the session context. This context is saved as the current context in the scriptstate. The script transitions to `checkAndEmit`.

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

$q =$ `startPIF` In this state, the script waits for a postMessage from the PIF containing a `ping` message. If such a message is received and the sender's window and origin match the PIF, the script sends a `pong` message back to the PIF and transitions to the state `runPIF`.

$q =$ `runPIF` This is the state in which `script_lpo_cif` interacts with the PIF. This state handles all postMessages the CIF expects to receive from the latest PIF (as recorded in `PIFindex` in its state). If the postMessage received was sent from the PIF's window and the PIF's origin, it behaves as follows, depending on the first element of the postMessage:

postMessage `beginProvisioning` The script responds with a postMessage to the PIF containing the email address of the identity which is to authenticate to the relying party (as recorded in the CIF's state).

postMessage `genKeyPair` The script creates a fresh key pair (i.e. the CIF chooses a fresh nonce) and sends the public key contained in a postMessage to the PIF.

postMessage `registerCertificate` The script stores the UC received in this postMessage in the CIF's state and transitions to the state `createCAPforRP`.

postMessage `raiseProvisioningFailure` This message indicates to the CIF that no one is logged in. This information is recorded in the CIF's state accordingly. The script transitions to the state `sendLogout` in which the CIF's parent window will be notified that no one is logged in.

$q =$ `createCAPforRP` In this state, the script creates an IA for the request origin (as recorded in the scriptstate), combines the IA with the UC to a CAP, and sends the CAP in a postMessage to its parent restricting the receiver to the request origin.

$q =$ `sendLogout` In this state, the script sends a `logout` postMessage to the parent document and then transitions to the `default` state.

We now specify the relation *script_lpo_cif* $\subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the CIF's scripting process formally.

Just like all scripts, as explained in Section 2.5 (see also Algorithm A.7 for the formal specification), the input term this script obtains from the browser contains the cleaned tree of the browser's windows and documents *tree*, the nonce of the current document *docnonce*, its own scriptstate *scriptstate* (as defined in Definition 79), a sequence of all inputs *scriptinput* (also containing already handled inputs), a dictionary *cookies* of all accessible cookies of the document's domain, the localStorage *localStorage* belonging to the document's origin, and the secrets *secrets* of the document's origin. The script returns a new scriptstate s', a new set of cookies *cookies*$'$, a new localStorage *localStorage*$'$, and a term *command* denoting a command to the browser.

The relation of the script *script_lpo_cif* is defined in Algorithm D.4.

login.persona.org Login Dialog Script (script_lpo_ld). This script models the LD contents. Its formal specification, presented next, follows the one presented above for *script_lpo_cif*.

Algorithm D.4: Relation of *script_lpo_cif*.

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

let $s' := scriptstate$
let $cookies' := cookies$
let $localStorage' := localStorage$
let $sessionStorage' := sessionStorage$
switch $s'.\texttt{q}$ **do**
    case `init`
        let $command := \langle \texttt{POSTMESSAGE}, \mathsf{PARENTWINDOW}(tree, docnonce), \hookrightarrow \langle \texttt{cifready}, \langle\rangle\rangle, \bot\rangle$
        let $s'.\texttt{q} := \texttt{default}$
        stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
    case `default`
        let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
        if $\pi_1(input) \equiv \texttt{POSTMESSAGE}$ **then**
            let $senderWindow := \pi_2(input)$
            let $senderOrigin := \pi_3(input)$
            let $m := \pi_4(input)$
            if $senderWindow \equiv \mathsf{PARENTWINDOW}(tree, docnonce)$ **then**
                switch m **do**
                    case $\langle \texttt{loaded}, id\rangle$
                        if $s'.\texttt{requestOrigin} \equiv \bot$ **then**
                            let $s'.\texttt{requestOrigin} := senderOrigin$
                        end if
                        let $s'.\texttt{loggedInUser} := id$
                        if $s'.\texttt{pause} \equiv \top$ **then**
                            stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
                        else if $s'.\texttt{context} \equiv \bot$ **then**
                            let $s'.\texttt{q} := \texttt{fetchContext}$
                            stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
                        else
                            let $s'.\texttt{q} := \texttt{checkAndEmit}$
                            stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
                        end if
                    case $\langle \texttt{dlgRun}, \langle\rangle\rangle$
                        let $s'.\texttt{pause} := \top$
                        stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
                    case $\langle \texttt{dlgCmplt}, \langle\rangle\rangle$
                        let $s'.\texttt{pause} := \bot$
                        let $s'.\texttt{q} := \texttt{fetchContext}$
                        stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
                    case $\langle \texttt{loggedInUser}, id\rangle$
                        let $s'.\texttt{loggedInUser} := id$
                        stop $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$

This algorithm is continued on the next page.

```
42:                     case ⟨logout, ⟨⟩⟩
43:                         if s′.requestOrigin ≡ ⊥ then
44:                             let s′.requestOrigin := senderOrigin
45:                         end if
46:                         let s′.loggedInUser := ⊥
47:                         remove the element with key s′.requestOrigin
                             ↪ from the dictionary localStorage′[siteInfo]
48:                         let s′.q := sendLogout
49:             end if
50:         end if
51:     case fetchContext
52:         let s′.refXHRctx := λ1
53:         let command := ⟨XMLHTTPREQUEST, URL^LPO_/ctx, GET, ⟨⟩, s′.refXHRctx⟩
54:         let s′.q := receiveContext
55:         stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
56:     case receiveContext
57:         let input, s′ := CHOOSEINPUT(s′, scriptinputs)
58:         if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s′.refXHRctx) then
59:             let s′.context := π2(input)
60:             let s′.q := checkAndEmit
61:             stop ⟨s′, cookies′, localStorage′, sessionStorage′, ⟨⟩⟩
62:         end if
63:     case checkAndEmit
64:         let s′.email := localStorage′[siteInfo][s′.requestOrigin]
65:         if (s′.email ≢ ⟨⟩)
             ↪ ∧(s′.loggedInUser ∉ {⟨⟩, ⊥} ⇒ (s′.loggedInUser ≢ s′.email))
             ↪ ∧(π1(s′.context) ≡ ⊤) then
66:             let s′.q := startPIF
67:             let url := ⟨URL, S, π2(s′.email), /pif⟩
68:             let s′.PIFindex := |subwindows| + 1
                    → Index of the next subwindow to be created.
69:             let command := ⟨IFRAME, url, _SELF⟩
70:             stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
71:         else if s′.loggedInUser ≡ ⟨⟩ then
72:             let s′.q := default
73:             stop ⟨s′, cookies′, localStorage′, sessionStorage′, ⟨⟩⟩
74:         else
75:             let s′.q := sendLogout
76:             stop ⟨s′, cookies′, localStorage′, sessionStorage′, ⟨⟩⟩
77:         end if
```

This algorithm is continued on the next page.

```
78:     case startPIF
79:         let idpOrigin := ⟨π2(s'.email), S⟩
80:         let input, s' := CHOOSEINPUT(s', scriptinputs)
81:         let pifNonce := π_{s'.PIFindex}(subwindows).nonce
82:         if π1(input) ≡ POSTMESSAGE then
83:             let senderWindow := π2(input)
84:             let senderOrigin := π3(input)
85:             let m := π4(input)
86:             if m ≡ ping ∧ senderWindow ≡ pifNonce
                ↪ ∧senderOrigin ≡ idpOrigin then
87:                 let command := ⟨POSTMESSAGE, pifNonce, pong, idpOrigin⟩
88:                 let s'.q := runPIF
89:                 stop ⟨s', cookies', localStorage', sessionStorage', command⟩
90:             end if
91:         end if
92:     case runPIF
93:         let idpOrigin := ⟨π2(s'.email), S⟩
94:         let input, s' := CHOOSEINPUT(s', scriptinputs)
95:         let pifNonce := π_{s'.PIFindex}(subwindows).nonce
96:         if π1(input) ≡ POSTMESSAGE then
97:             let senderWindow := π2(input)
98:             let senderOrigin := π3(input)
99:             let m := π4(input)
100:            if senderWindow ≡ pifNonce ∧ senderOrigin ≡ idpOrigin then
101:                switch π1(m) do
102:                    case beginProvisioning
103:                        let jschannel_nonce := π2(m)
104:                        let command := ⟨POSTMESSAGE, pifNonce,
                            ↪ ⟨jschannel_nonce, s'.email⟩, idpOrigin⟩
105:                        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
106:                    case genKeyPair
107:                        let jschannel_nonce := π2(m)
108:                        let s'.key := λ2
109:                        let command := ⟨POSTMESSAGE, pifNonce,
                            ↪ ⟨jschannel_nonce, pub(s'.key)⟩, idpOrigin⟩
110:                        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
111:                    case registerCertificate
112:                        if π1(extractmsg(π2(m))) ≡ s'.email ∧ s'.email ≢ ⟨⟩ then
                            → This check is our fix against identity injection.
113:                            let s'.uc := π2(m)
114:                            let s'.q := createCAPforRP
115:                        end if
116:                        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
117:                    case raiseProvisioningFailure
118:                        let s'.loggedInUser := ⊥
119:                        let s'.q := sendLogout
120:                        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
121:            end if
122:        end if
```

This algorithm is continued on the next page.

```
123:    case createCAPforRP
124:        let ia := sig(s'.requestOrigin, s'.key)
125:        let cap := ⟨s'.uc, ia⟩
126:        let command := ⟨POSTMESSAGE, PARENTWINDOW(tree, docnonce),
            ↪ ⟨login, cap⟩, s'.requestOrigin⟩
127:        let s'.q := null
128:        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
129:    case sendLogout
130:        let command := ⟨POSTMESSAGE, PARENTWINDOW(tree, docnonce),
            ↪ ⟨logout, ⟨⟩⟩, ⊥⟩
131:        let s'.q := default
132:        stop ⟨s', cookies', localStorage', sessionStorage', command⟩
133: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
```

Definition 80. A *scriptstate s of script_lpo_ld* is a term of the form

$$\langle q, requestOrigin, context, email, key, uc, handledInputs, refXHRctx, refXHRLPOauth, PIFindex\rangle$$

with $q \in \mathbb{S}$, $requestOrigin \in \mathsf{Origins} \cup \{\bot\}$, $context \in \mathcal{T}_{\mathcal{N}}$, $email \in \mathsf{ID} \cup \{\bot\}$, $key \in \mathcal{N} \cup \{\bot\}$, $uc \in \mathcal{T}_{\mathcal{N}}$, $handledInputs \subset^{\langle\rangle} \mathbb{N}$, $refXHRctx, refXHRLPOauth \in \mathcal{N} \cup \{\bot\}$, $PIFindex \in \mathbb{N} \cup \{\bot\}$. The *initial scriptstate* $initState_{ld}$ is the state $\langle \mathtt{init}, \bot, \bot, \bot, \bot, \bot, \langle\rangle, \bot, \bot, \bot\rangle$. ◇

We formally specify the relation $script\_lpo\_ld \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the LD's scripting process in Algorithm D.5.

Relying Party Web Page Script (script_rp_index). This script models the default Web page at a RP. The user usually triggers the login process on this page. In reality, this page has its own script(s) and includes a script from LPO. In our model, we combine both scripts into *script_rp_index*. In particular, this script is responsible for creating the CIF and the LD iframes/subwindows, whose contents are loaded from LPO.

Definition 81. A *scriptstate s of script_rp_index* is a term of the form

$$\langle q, CIFindex, LDindex, dialogRunning, cap, handledInputs, refXHRcap\rangle$$

with $q \in \mathbb{S}$, $CIFindex \in \mathbb{N} \cup \{\bot\}$, $dialogRunning \in \{\top, \bot\}$, $cap \in \mathcal{T}_{\mathcal{N}}$, $handledInputs \subset^{\langle\rangle} \mathbb{N}$, $refXHRcap \in \mathcal{N} \cup \{\bot\}$. We call s the *initial scriptstate of script_rp_index* iff

$$s \equiv \langle \mathtt{init}, \bot, \bot, \bot, \langle\rangle, \langle\rangle, \bot\rangle\ .$$

◇

Algorithm D.5: Relation of *script_lpo_ld*.

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

let $s' := scriptstate$
let $cookies' := cookies$
let $localStorage' := localStorage$
let $sessionStorage' := sessionStorage$
switch $s'.\texttt{q}$ **do**
  case `init`
    let $command := \langle \texttt{POSTMESSAGE}, \mathsf{OPENERWINDOW}(tree, docnonce), \langle \texttt{ldready}, \langle\rangle\rangle, \bot\rangle$
    let $s'.\texttt{q} := \texttt{start}$
    stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
  case `start`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $\pi_1(input) \equiv \texttt{POSTMESSAGE}$ **then**
      let $senderWindow := \pi_2(input)$
      let $senderOrigin := \pi_3(input)$
      let $m := \pi_4(input)$
      if $m \equiv \langle \texttt{request}, \langle\rangle\rangle \wedge senderWindow \equiv \mathsf{OPENERWINDOW}(tree, docnonce)$ **then**
        let $s'.\texttt{requestOrigin} := senderOrigin$
        let $s'.\texttt{refXHRctx} := \lambda_1$
        let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{\mathsf{LPO}}_{/\texttt{ctx}}, \texttt{GET}, \langle\rangle, s'.\texttt{refXHRctx}\rangle$
        let $s'.\texttt{q} := \texttt{receiveContext}$
        stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
      end if
    end if
  case `receiveContext`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $(\pi_1(input) \equiv \texttt{XMLHTTPREQUEST}) \wedge (\pi_3(input) \equiv s'.\texttt{refXHRctx})$ **then**
      let $s'.\texttt{context} := \pi_2(input)$
      let $s'.\texttt{q} := \texttt{startPIF}$
      let $idpnonce := sessionStorage[\texttt{idpnonce}]$
      if $idpnonce \equiv \langle\rangle \vee localStorage[idpnonce] \equiv \langle\rangle$ **then**
        let $s'.\texttt{email} \leftarrow ids$
      else
        let $s'.\texttt{email} := localStorage[idpnonce]$
        let $sessionStorage[\texttt{idpnonce}] := \langle\rangle$
      end if
      let $url := \langle \texttt{URL}, \texttt{S}, \pi_2(s'.\texttt{email}), /\texttt{pif}\rangle$
      let $s'.\texttt{PIFindex} := |subwindows| + 1$
        → Index of the next subwindow to be created.
      let $command := \langle \texttt{IFRAME}, url, \texttt{\_SELF}\rangle$
      stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
    end if

This algorithm is continued on the next page.

```
41:     case startPIF
42:         let idpOrigin := ⟨π2(s'.email), S⟩
43:         let input, s' := CHOOSEINPUT(s', scriptinputs)
44:         let pifNonce := π_{s'.PIFindex}(subwindows).nonce
45:         if π1(input) ≡ POSTMESSAGE then
46:             let senderWindow := π2(input)
47:             let senderOrigin := π3(input)
48:             let m := π4(input)
49:             if m ≡ ping ∧ senderWindow ≡ pifNonce
                  ↪ ∧senderOrigin ≡ idpOrigin then
50:                 let command := ⟨POSTMESSAGE, pifNonce, pong, idpOrigin⟩
51:                 let s'.q := runPIF
52:                 stop ⟨s', cookies', localStorage', sessionStorage', command⟩
53:             end if
54:         end if
55:     case runPIF
56:         let idpOrigin := ⟨π2(s'.email), S⟩
57:         let input, s' := CHOOSEINPUT(s', scriptinputs)
58:         let pifNonce := π_{s'.PIFindex}(subwindows).nonce
59:         if π1(input) ≡ POSTMESSAGE then
60:             let senderWindow := π2(input)
61:             let senderOrigin := π3(input)
62:             let m := π4(input)
63:             if senderWindow ≡ pifNonce ∧ senderOrigin ≡ idpOrigin then
64:                 switch π1(m) do
65:                     case beginProvisioning
66:                         let jschannel_nonce := π2(m)
67:                         let command := ⟨POSTMESSAGE, pifNonce,
                              ↪ ⟨jschannel_nonce, s'.email⟩, idpOrigin⟩
68:                         stop ⟨s', cookies', localStorage', sessionStorage', command⟩
69:                     case genKeyPair
70:                         let jschannel_nonce := π2(m)
71:                         let s'.key := λ2
72:                         let command := ⟨POSTMESSAGE, pifNonce,
                              ↪ ⟨jschannel_nonce, pub(s'.key)⟩, idpOrigin⟩
73:                         stop ⟨s', cookies', localStorage', sessionStorage', command⟩
74:                     case registerCertificate
75:                         if π1(extractmsg(π2(m))) ≡ s'.email ∧ s'.email ≢ ⟨⟩ then
                              → This check is our fix against identity injection.
76:                             let s'.uc := π2(m)
77:                             let loggedIn := π1(s'.context)
78:                             if loggedIn ≡ ⊤ then
79:                                 let s'.q := createCAPforRP
80:                             end if
81:                             let s'.q := createCAPforLPO
82:                         end if
83:                         stop ⟨s', cookies', localStorage', sessionStorage', command⟩
```

This algorithm is continued on the next page.

```
84:                     case raiseProvisioningFailure
85:                         let idpnonce := λ3
86:                         let localStorage′[idpnonoce] := s′.email
87:                         let sessionStorage′[idpnonce] := idpnonce
88:                         let command := ⟨HREF, ⟨URL, S, π2(s′.email), ⟨⟩⟩, _SELF⟩
89:                         stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
90:             end if
91:         end if
92:     case createCAPforLPO
93:         let ia := sig(⟨dom(LPO), S⟩, s′.key)
94:         let cap := ⟨s′.uc, ia⟩
95:         let body := ⟨cap, π2(s′.context)⟩
96:         let s′.refXHRLPOauth := λ4
97:         let command := ⟨XMLHTTPREQUEST, URL^LPO_/auth, POST, body, s′.refXHRLPOauth⟩
98:         let s′.q := receiveLPOauthresponse
99:         stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
100:    case receiveLPOauthresponse
101:        let input, s′ := CHOOSEINPUT(s′, scriptinputs)
102:        if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s′.refXHRLPOauth)
            ↪ ∧π2(input) ≡ ⊤ then
103:            let π1(s′.context) := ⊤
104:            let s′.q := createCAPforRP
105:            stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
106:        end if
107:    case createCAPforRP
108:        let ia := sig(s′.requestOrigin, s′.key)
109:        let cap := ⟨s′.uc, ia⟩
110:        let command := ⟨POSTMESSAGE, OPENERWINDOW(tree, docnonce),
            ↪ ⟨response, cap⟩, s′.requestOrigin⟩
111:        let s′.q := null
112:        let localStorage′[siteInfo][s′.requestOrigin] := s′.email
113:        stop ⟨s′, cookies′, localStorage′, sessionStorage′, command⟩
114: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
```

We formally specify the relation *script_rp_index* $\subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the RP-Doc's scripting process in Algorithm D.6.

In Lines 7–11 and 33–37 the script asks the browser to create iframes. To obtain the window reference for these iframes, the script first determines the current number of subwindows and stores it (incremented by 1) in the scriptstate (`CIFindex` and `LDindex`, respectively). When the script is invoked the next time, the iframe the script asked to be created will have been added to the sequence of subwindows by the browser directly following the previously existing subwindows. The script can therefore access the iframe by the indexes `CIFindex` and `LDindex`, respectively.

Identity Provider Authentication Dialog Script (script_idp_ad).

Note that *script_idp_ad* does not read or change its scriptstate. Hence, we omit the definition of the scriptstate for this script. We define the relation of the script in Algorithm D.7.

Identity Provider Provisioning Iframe Script (script_idp_pif). The set of scriptstates of this script is defined as follows:

Definition 82. A *scriptstate s of script_idp_pif* is a term of the form $\langle q, \mathit{emails}, \mathit{pubkeys}, \mathit{ucs}, \mathit{provisioningnonces}, \mathit{genkeypairnonces}, \mathit{xhrnonces}, \mathit{handledInputs}\rangle$ with $q \in \mathbb{S}$, *emails*, *pubkeys*, $\mathit{ucs} \in \mathcal{T}_{\mathcal{N}}$, *provisioningnonces*, *genkeypairnonces*, *xhrnonces* $\in \mathcal{N} \cup \{\bot\}$, *handledInputs* $\subset^{\langle\rangle} \mathbb{N}$. We call *s* the *initial scriptstate of script_idp_pif* iff $s \equiv \langle \texttt{init}, \langle\rangle, \langle\rangle, \langle\rangle, \bot, \bot, \bot\rangle$. ◇

We define the script's relation in Algorithm D.8

D.3.10. Important Events

As described in Section 3.2, we define events that (if they occur in a certain processing step) specify important actions of our SSO protocol.

Definition 83 (Refinement of Definition 5 for BrowserID in Primary Mode: Start of an SSO Flow). Let $\mathcal{BID}^p$ be an BrowserID primary mode Web system. Let ρ be a run of $\mathcal{BID}^p$. Let $Q \in \rho$ be a processing step, *b* a browser, and *r* an RP. We write $\mathsf{started}(Q, b, r)$ iff in *Q*, the browser *b* is triggered and selects to run the script of a document and this script is `script_rp_index` and the document is stored in *b* under a secure origin of *r* and —when executing the script— in Line 28 of Algorithm D.6 the value `openLD` is (non-deterministically) chosen. ◇

Definition 84 (Refinement of Definition 6 for BrowserID in Primary Mode: Selection of an IdP). Let $\mathcal{BID}^p$ be an BrowserID primary mode Web system. Let ρ be a run of $\mathcal{BID}^p$. Let $Q \in \rho$ be a processing step, *b* a browser, *i* an IdP, and *id* some identity with $\mathsf{governor}(id) = i$.

Algorithm D.6: Relation of *script_rp_index*.

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

let $s' := scriptstate$
let $cookies' := cookies$
let $localStorage' := localStorage$
let $sessionStorage' := sessionStorage$
switch $s'.\mathtt{q}$ **do**
  case `init`
    let $command := \langle \mathtt{IFRAME}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{cif}}, \mathsf{GETWINDOW}(tree, docnonce)\rangle$
    let $s'.\mathtt{q} := \mathtt{receiveCIFReady}$
    let $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
    let $s'.\mathtt{CIFindex} := |subwindows| + 1$ → Index of the next subwindow to be created.
    stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
  case `receiveCIFReady`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $\pi_1(input) \equiv \mathtt{POSTMESSAGE}$ **then**
      let $senderWindow := \pi_2(input)$
      let $senderOrigin := \pi_3(input)$
      let $m := \pi_4(input)$
      let $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
      if $(m \equiv \langle \mathtt{cifready}, \langle\rangle\rangle)$
        ↪ $\wedge(senderOrigin \equiv \mathsf{origin}_{\mathsf{LPO}})$
        ↪ $\wedge(senderWindow \equiv \pi_{s'.\mathtt{CIFindex}}(subwindows).\mathtt{nonce})$ **then**
        let $id \leftarrow \{\bot, \langle\rangle\} \cup \mathsf{ID}$
        let $command := \langle \mathtt{POSTMESSAGE}, \pi_{s'.\mathtt{CIFindex}}(subwindows),$
          ↪ $\langle \mathtt{loaded}, id\rangle, \mathsf{origin}_{\mathsf{LPO}}\rangle$
        let $s'.\mathtt{q} := \mathtt{default}$
        stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$
      end if
    end if
  case `default`
    if $s'.\mathtt{dialogRunning} \equiv \bot$ **then**
      let $choice \leftarrow \{\mathtt{openLD}, \mathtt{handlePM}\}$
    else
      let $choice := \mathtt{handlePM}$
    end if
    if $choice \equiv \mathtt{openLD}$ **then**
      let $s'.\mathtt{dialogRunning} := \top$
      let $command := \langle \mathtt{HREF}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{ld}}, \mathtt{\_BLANK}\rangle$
      let $s'.\mathtt{ldpointer} := w$ with w being a pointer to the next top-level window to be created
        ↪ (extracted from *tree*)
      let $s'.\mathtt{q} := \mathtt{default}$
      stop $\langle s', cookies', localStorage', sessionStorage', command\rangle$

This algorithm is continued on the next page.

38: **else**
39: **let** $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
40: **if** $\pi_1(input) \equiv \texttt{POSTMESSAGE}$ **then**
41: **let** $senderWindow := \pi_2(input)$
42: **let** $senderOrigin := \pi_3(input)$
43: **let** $m := \pi_4(input)$
44: **let** $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
45: **if** $senderOrigin \equiv \mathsf{origin}_{\mathsf{LPO}}$ **then**
46: **if** $senderWindow \equiv \pi_{s'.\texttt{CIFindex}}(subwindows).\texttt{nonce}$ **then**
47: **if** $\pi_1(m) \equiv \texttt{login}$ **then**
48: **let** $s'.\texttt{cap} := \pi_2(m)$
49: **let** $s'.\texttt{q} := \texttt{sendCAP}$
50: **stop** $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
51: **else if** $\pi_1(m) \equiv \texttt{logout}$ **then**
52: **let** $s'.\texttt{q} := \texttt{default}$
53: **stop** $\langle s', cookies', localStorage', sessionStorage', \langle\rangle\rangle$
54: **end if**
55: **else if** $s'.\texttt{dialogRunning} \equiv \top \wedge senderWindow \equiv w$ with w being the nonce of the window pointed to by $s'.\texttt{ldpointer}$ **then**
56: **if** $\pi_1(m) \equiv \texttt{ldready}$ **then**
57: **let** $command := \langle \texttt{POSTMESSAGE}, \mathsf{AUXWINDOW}(tree, docnonce), \langle \texttt{request}, \langle\rangle\rangle, \mathsf{origin}_{\mathsf{LPO}}\rangle$
58: **let** $s'.\texttt{q} := \texttt{default}$
59: **stop** $\langle s', cookies', localStorage', sessionStorage', command\rangle$
60: **else if** $\pi_1(m) \equiv \texttt{response}$ **then**
61: **let** $s'.\texttt{dialogRunning} := \bot$
62: **let** $s'.\texttt{cap} := \pi_2(m)$
63: **let** $command := \langle \texttt{CLOSE}, \mathsf{AUXWINDOW}(tree, docnonce)\rangle$
64: **let** $s'.\texttt{q} := \texttt{dlgClosed}$
65: **stop** $\langle s', cookies', localStorage', sessionStorage', command\rangle$
66: **end if**
67: **end if**
68: **end if**
69: **end if**
70: **end if**
71: **case** `dlgClosed`
72: **let** $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
73: **let** $id := \pi_1(\mathsf{extractmsg}(\pi_1(s'.\texttt{cap})))$ → Extract id from CAP.
74: **let** $command := \langle \texttt{POSTMESSAGE}, \pi_{s'.\texttt{CIFindex}}(subwindows).\texttt{nonce}, \langle \texttt{loggedInUser}, id\rangle, \mathsf{origin}_{\mathsf{LPO}}\rangle$
75: **let** $s'.\texttt{q} := \texttt{loggedInUser}$
76: **stop** $\langle s', cookies', localStorage', sessionStorage', command\rangle$
77: **case** `loggedInUser`
78: **let** $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
79: **let** $command := \langle \texttt{POSTMESSAGE}, \pi_{s'.\texttt{CIFindex}}(subwindows).\texttt{nonce}, \langle \texttt{dlgCmplt}, \langle\rangle\rangle, \mathsf{origin}_{\mathsf{LPO}}\rangle$
80: **let** $s'.\texttt{q} := \texttt{sendCAP}$
81: **stop** $\langle s', cookies', localStorage', sessionStorage', command\rangle$

This algorithm is continued on the next page.

```
82:     case sendCAP
83:         let s'.refXHRcap := λ1
84:         let host, protocol such that
              ↪ ⟨host, protocol⟩ ≡ GETORIGIN(tree, docnonce)
              ↪ if possible; otherwise stop
              ↪ ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
85:         let command := ⟨XMLHTTPREQUEST, ⟨URL, protocol, host, /, ⟨⟩⟩, POST, s'.cap,
              ↪ s'.refXHRcap⟩    → Relay received CAP to RP.
86:         let s'.q := receiveServiceToken
87:         stop ⟨s', cookies', localStorage', sessionStorage', command⟩
88:     case receiveServiceToken
89:         let input, s' := CHOOSEINPUT(s', scriptinputs)
90:         if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRcap) then
91:             let s'.q := default
92:             stop ⟨s', cookies', localStorage', sessionStorage', ⟨⟩⟩
93:         end if
94: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
```

Algorithm D.7: Relation of *script_idp_ad*.

```
Input: ⟨tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets⟩
 1: let action ← {authenticate, navigate}
 2: if action ≡ authenticate then
 3:     let email ← ids
 4:     let secret ← secrets
 5:     let body := ⟨email, secret⟩
 6:     let host, protocol such that
          ↪ ⟨host, protocol⟩ ≡ GETORIGIN(tree, docnonce)
          ↪ if possible; otherwise
          ↪ stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
 7:     let command := ⟨XMLHTTPREQUEST, ⟨URL, protocol, host, /auth, ⟨⟩⟩, POST, body, ⊥⟩
 8:     stop ⟨scriptstate, cookies', localStorage', sessionStorage', command⟩
 9: else
10:     let command := ⟨HREF, ⟨URL, S, dom(LPO), /ld, ⟨⟩⟩, _SELF⟩
11:     stop ⟨scriptstate, cookies', localStorage', sessionStorage', command⟩
12: end if
```

Algorithm D.8: Relation of *script_idp_pif*.

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

let $s' := scriptstate$
switch s'.q **do**
  case `init`
    let $command := \langle$`POSTMESSAGE`, PARENTWINDOW($tree$,$docnonce$), ↪ $\langle$`ping`,$\langle\rangle\rangle$, $\langle$dom(LPO), S$\rangle\rangle$
    let s'.q := `waiting`
    stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
  case `waiting`
    let $input, s' :=$ CHOOSEINPUT($s', scriptinputs$)
    let $senderWindow := \pi_2(input)$
    let $senderOrigin := \pi_3(input)$
    let $m := \pi_4(input)$
    if $\pi_1(input) \in \{$`ping`,`pong`$\}$ ↪ $\wedge senderWindow \equiv$ PARENTWINDOW($tree$,$docnonce$) ↪ $\wedge senderOrigin \equiv \langle$dom(LPO),S$\rangle$ **then**
      let s'.q := `default`
    end if
    stop $\langle s', cookies, localStorage, sessionStorage, \langle\rangle\rangle$
  case `default`
    let $action \leftarrow \{$`beginprovisioning`, `genkeypair`, `registercert`, ↪ `raisefailure`, `requestuc`, `handleresponse`$\}$
    switch $action$ **do**
      case `beginprovisioning`
        let $jschannel\_nonce := \lambda_1$
        let $command := \langle$`POSTMESSAGE`, PARENTWINDOW($tree$,$docnonce$), ↪ $\langle$`beginProvisioning`,$jschannel\_nonce\rangle$, dom(LPO)$\rangle$
        let s'.`provisioningnonces` := ↪ s'.`provisioningnonces` $+^{\langle\rangle}$ $jschannel\_nonce$
        stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
      case `genkeypair`
        let $jschannel\_nonce := \lambda_1$
        let $command := \langle$`POSTMESSAGE`, PARENTWINDOW($tree$,$docnonce$), ↪ $\langle$`genKeyPair`,$jschannel\_nonce\rangle$, dom(LPO)$\rangle$
        let s'.`genkeypairnonces` := ↪ s'.`genkeypairnonces` $+^{\langle\rangle}$ $jschannel\_nonce$
        stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
      case `registercert`
        if s'.`ucs` $\not\equiv \langle\rangle$ **then**
          let $uc \leftarrow s'$.`ucs`
          let $command :=$ $\langle$`POSTMESSAGE`,PARENTWINDOW($tree$,$docnonce$), ↪ $\langle$`registerCertificate`,$uc\rangle$, dom(LPO)$\rangle$
          stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
        end if

This algorithm is continued on the next page.

```
35:             case raisefailure
36:                 let command := ⟨POSTMESSAGE, PARENTWINDOW(tree,docnonce),
                      ↪ ⟨raiseProvisioningFailure, ⊥⟩, dom(LPO)⟩
37:                 stop ⟨s′,cookies,localStorage,sessionStorage,command⟩
38:             case requestuc
39:                 if s′.emails ≢ ⟨⟩ ∧ s′.pubkeys ≢ ⟨⟩ then
40:                     let email ← s′.emails
41:                     let pubkey ← s′.pubkeys
42:                     let body := ⟨email,pubkey⟩
43:                     let xhrnonce := λ2
44:                     let s′.xhrnonces := s′.xhrnonces +⟨⟩ xhrnonce
45:                     let host,protocol such that
                          ↪ ⟨host,protocol⟩ ≡ GETORIGIN(tree,docnonce)
                          ↪ if possible; otherwise
                          ↪ stop ⟨s′,cookies,localStorage,sessionStorage,⟨⟩⟩
46:                     let command := ⟨XMLHTTPREQUEST,
                          ↪ ⟨URL,protocol,host,/certreq,⟨⟩⟩,POST,body,xhrnonce⟩
47:                     stop ⟨s′,cookies,localStorage,sessionStorage,command⟩
48:                 end if
49:             case handleresponse
50:                 let input,s′ := CHOOSEINPUT(s′,scriptinputs)
51:                 if π1(input) ≡ POSTMESSAGE then
52:                     let senderWindow := π2(input)
53:                     let senderOrigin := π3(input)
54:                     let m := π4(input)
55:                     if senderWindow ≡ PARENTWINDOW(tree,docnonce)
                          ↪ ∧senderOrigin ≡ ⟨dom(LPO),S⟩ then
56:                         if π1(m) ∈ s′.provisioningnonces then
57:                             let s′.emails := s′.emails +⟨⟩ π2(m)
58:                         else if π1(m) ∈ s′.genkeypairnonces then
59:                             let s′.pubkeys := s′.pubkeys +⟨⟩ π2(m)
60:                         end if
61:                         stop ⟨s′,cookies,localStorage,sessionStorage,⟨⟩⟩
62:                     end if
63:                 else if π1(input) ≡ XMLHTTPREQUEST
                      ↪ ∧π3(input) ∈ s′.xhrnonces then
64:                     let s′.ucs := s′.ucs +⟨⟩ π2(input)
65:                     stop ⟨s′,cookies,localStorage,sessionStorage,⟨⟩⟩
66:                 end if
67: stop ⟨scriptstate,cookies,localStorage,sessionStorage,⟨⟩⟩
```

We write $\mathsf{selectedIdP}(Q,b,i)$ iff in Q, the browser b is triggered and selects to run the script `script_lpo_ld` in some document and — in that script — in Lines 31ff. of Algorithm D.5, *id* is selected from the browser's identities. ◇

Definition 85 (Refinement of Definition 7 for BrowserID in Primary Mode: Selection of an Identity). Let $\mathcal{BID}^p$ be an BrowserID primary mode Web system. Let ρ be a run of $\mathcal{BID}^p$. Let $Q \in \rho$ be a processing step, b a browser, i an IdP, and *id* an identity. We write $\mathsf{selectedID}(Q,b,i,id)$ iff in Q, the browser b is triggered and selects to run (in some document) the script `script_lpo_ld` and — in that script — in Lines 31ff. of Algorithm D.5, *id* is selected from the browser's identities. ◇

Definition 86 (Refinement of Definition 8 for BrowserID in Primary Mode: User is Logged in at RP). Let $\mathcal{BID}^p$ be an BrowserID primary mode Web system. Let ρ be a run of $\mathcal{BID}^p$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, *id* an identity, and n some term. We write $\mathsf{loggedIn}(Q,b,r,id,n)$ iff in Q, b is triggered, selects a document containing the script `script_rp_index` under a secure origin of r and –in that script— in Lines 90ff. of Algorithm D.6 a response to an XHR is processed that contains n in its body, while in the state of r (before and after Q — note that the state of r cannot be altered during Q), for the subterm *serviceTokens*, we have $serviceTokens[n] \equiv id$. ◇

D.4. Proof of Theorem 1 (Security w.r.t. Authentication)

In order to prove Theorem 1, we have to prove that the BrowserID primary mode Web system satisfies security w.r.t. authentication (Definition 12).

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

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

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

CAP, and *sts* is the Strict-Transport-Security header (as in the definition of RP's relation):

$$req_{\text{cap}} = \mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, n_{\text{cap}}, \mathtt{POST}, d_r, path, params, [\mathtt{Origin} : \langle d_r, \mathtt{S}\rangle], c\rangle, k\rangle, \mathsf{pub}(\mathsf{tlskey}(d_r))) \tag{D.1}$$

$$resp_{\text{cap}} = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n_{\text{cap}}, 200, \langle sts\rangle, \langle n, i\rangle\rangle, k) \tag{D.2}$$

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

$$s_{m-1} \xrightarrow[r \to \{(x:a_r:resp_{\text{cap}})\}]{(a_r:x:req_{\text{cap}}) \to r} s_m \ .$$

From the assumption and the definition of RPs it follows that c is of the following form:

$$\begin{aligned} c &= \langle uc, ia\rangle \\ &\equiv \langle \mathsf{sig}(\langle i, \mathsf{pub}(k_u)\rangle, k_{\text{sign}}), \mathsf{sig}(\langle d_r, \mathtt{S}\rangle, k_u)\rangle \end{aligned}$$

where k_u and k_{sign} are some private keys. When we write $i = \langle i_{\text{name}}, i_{\text{domain}}\rangle$, we have that:

$$c \equiv \langle \mathsf{sig}(\langle\langle i_{\text{name}}, i_{\text{domain}}\rangle, \mathsf{pub}(k_u)\rangle, k_{\text{sign}}), \mathsf{sig}(\langle d_r, \mathtt{S}\rangle, k_u)\rangle \ .$$

As r accepts the CAP c, we know that $\mathsf{pub}(k_{\text{sign}}) \equiv S_j(r).\mathtt{signkeys}[i_{\text{domain}}]$. As the subterm `signkeys` of r's state is never changed, we have $S_j(r).\mathtt{signkeys} = S_0(r).\mathtt{signkeys}$. With the definition of the initial state of r (See Definition 78), we have that $\mathsf{pub}(k_{\text{sign}}) \equiv S_j(r).\mathtt{signkeys}[i_{\text{domain}}] \equiv \mathsf{pub}(\mathsf{signkey}(\mathsf{dom}^{-1}(i_{\text{domain}})))$.

The private key $\mathsf{signkey}(\mathsf{dom}^{-1}(i_{\text{domain}}))$ is initially only known to the DY process $idp := \mathsf{dom}^{-1}(i_{\text{domain}}) = \mathsf{governor}(i)$. From the assumption we know that *idp* is an honest IdP (and not the attacker, a corrupted IdP, or some other DY process). As we can see in Algorithm D.2 (that defines the behavior of IdPs), the *signkey* can only be used in Line 4 and in Line 32. We know that Line 4 cannot be invoked as long as *idp* is honest, which it is in s_j and ever since s_0. For Line 32, we see that the key is not sent out to other processes. In s_j, the key can therefore not have been leaked to any other DY processes.

Knowing that in or before s_j, only *idp* can derive k_{sign} from its knowledge, it is easy to see that only *idp* can derive $\mathsf{sig}(x, k_{\text{sign}})$ for any x, and in particular, *uc*.

Now we want to see exactly how *idp* creates *uc* and which data it uses in this process.

We have already seen that *idp* creates the *uc* in Line 32 of Algorithm D.2. There may be more than one processing step in ρ where *idp* outputs *uc*.

Lemma 4. For all processing steps of the form

$$s_{\beta-1} \xrightarrow[idp\to\{(x:a_{idp}:resp_{\text{uc}})\}]{(a_{idp}:x:req_{\text{uc}})\to idp} s_{\beta} \tag{D.3}$$

(for some addresses x, a_{idp} with $\beta < j$, where $resp_{\text{uc}}$ is an encrypted HTTP response with the body $\langle uc \rangle$) it holds that req_{uc} was emitted by b.

Proof. To reach Line 32 of Algorithm D.2, several conditions have to be met for req_{uc}: It must be an encrypted HTTPS POST request with the path `/certreq`. The body of req_{uc} must be congruent to $\langle i, \mathsf{pub}(k_u) \rangle$. The request must contain a cookie with the name `sessionid` and some value *sessionid*. This value must be a valid key for the dictionary $s'.\texttt{sessions}$ and

$$i \in^{\langle\rangle} s'.\texttt{sessions}[sessionid] \; . \tag{D.4}$$

Initially, $s'.\texttt{sessions}$ is empty. It is only populated in Line 22 of Algorithm D.2. This line must have been executed in a previous processing step of the following form:

$$s_{\alpha-1} \xrightarrow[idp\to\{(x:a_{idp}:resp_{\text{auth}})\}]{(a_{idp}:x:req_{\text{auth}})\to idp} s_{\alpha} \tag{D.5}$$

(for some addresses x, a_{idp} with $s_{\alpha} < s_{\beta}$). In this step, $s'.\texttt{sessions}$ was populated with a new entry for the session id *sessionid*.

From Algorithm D.2 we can see that req_{auth} must meet the following conditions: It must be an HTTPS POST request, must contain a specific Origin header and its body must contain a pair $\langle i_{\text{in}}, secret_{\text{in}} \rangle$ such that the id/password combination matches a combination stored in $S_{\alpha-1}(idp).\texttt{users}$. As we have that $S_{\alpha-1}(idp).\texttt{users} = S_0(idp).\texttt{users}$ and with the initial definition

$$S_0(idp).\texttt{users} = \langle \{ \langle s, \langle \mathsf{IDsofSecret}(s) \rangle \rangle | \mathsf{Secrets}^i \} \rangle \tag{D.6}$$

we can see that $i_{\text{in}} \in \mathsf{IDsofSecret}(secret_{\text{in}})$. As the list of authenticated ids in the session is then (in Line 22 of Algorithm D.2) populated with $\mathsf{IDsofSecret}(secret_{\text{in}})$ and with (D.4) we have that $i \in \mathsf{IDsofSecret}(secret_{\text{in}})$. Now, $\mathsf{IDsofSecret}$ assigns the ids to their secrets according to $\mathsf{secretOfID}$, i.e., it must hold that

$$\mathsf{secretOfID}(i) = secret_{\text{in}} \; . \tag{D.7}$$

This secret can be owned by at most one browser, and according to the definitions of the initial

knowledge of the DY processes in D.3, it is initially only known to the owner of the secret $\mathsf{ownerOfSecret}(\mathit{secret}_{\text{in}})$ (see Section D.3.5) and to one specific IdP (see Section D.3.7), in this case $i_{\text{domain}} \in \mathsf{dom}(\mathit{idp})$ (because otherwise, *idp* would not accept this id).

From Algorithm D.2 we can see that the IdP never uses this secret to create messages as long as it is honest, which it is by precondition.

With (D.7) we see that initially, only $\mathsf{ownerOfSecret}(\mathsf{secretOfID}(i)) = \mathsf{ownerOfID}(i)$ knows the secret $\mathit{secret}_{\text{in}}$, which, by assumption, is not fully corrupted in s_j, and thus, with the request order given for (D.3) and (D.5) is not fully corrupted in s_α. (Once fully corrupted, browsers stay fully corrupted.)

(*): Honest browsers release secrets only to scripts that are loaded from a specific origin. In this case, according to the initial state given in Section D.3.5, the secret $\mathsf{secretOfID}(i)$ is only released to scripts from the origin $\langle i_{\text{domain}}, \mathtt{S} \rangle$. For any such script (or document), with Lemma 2 and the definition of the browser's key mapping in Section D.3.5, we can see that any script that has access to the secret was sent by *idp*. This DY process is also the governor of *i*, which is, by assumption, not corrupted. Therefore, *idp* can only deliver either the script *script_idp_pif* or the script *script_idp_ad*. We can now check, that both scripts, running in a browser, never send this secret to any other DY process than *idp*, and trigger only encrypted requests to do so.

In *script_idp_pif* (Algorithm D.8), the subterm *secret* of the state is not used at all; therefore, the script triggers no outgoing message containing the secret at all.

In *script_idp_ad* (Algorithm D.7), *secret* is only used as a part of an HTTP request to the document's own origin (which therefore is the origin for which the secret is stored in the browser's list of secrets, which therefore must be $\langle i_{\text{domain}}, \mathtt{S} \rangle$). The request's data is not stored in the scriptstate.

We now know that all entities that have access to *secret* (the browser *b* and the IdP *idp*) never leak it. As *idp* never creates any HTTP(S) requests, *b* must have created $\mathit{req}_{\text{auth}}$ before the processing step $s_{\alpha-1} \rightarrow s_\alpha$.

In this processing step, *idp* creates a new session id (*sessionid*). This id is sent out only once (in Line 25 of Algorithm D.2), which, in our case, is $\mathit{resp}_{\text{auth}}$. With Corollary 1 we can see that from this (encrypted) response $\mathit{resp}_{\text{auth}}$, only *b* can derive the contents, especially the contents of the `Set-Cookie` header. As in *b*, the cookie is stored as a *secure*, *HTTP only* cookie, *b* releases the contents of this cookie only as a `Cookie` header to the origin $\langle i_{\text{domain}}, \mathtt{S} \rangle$. Given the keymapping in *b*'s state, requests to this origin are handled by *idp*, and with Algorithm D.2 it is easy to see that the `Cookie` header is only used for validating the UC request, but is not used anywhere else. All in all, *b* and *idp* do not leak the session id *sessionid*.

As *sessionid* is an important part of req_{uc}, we can see that this request must have been emitted by *b*. □

Lemma 5. The secret key k_u was chosen by the browser b from its own nonces, i.e., $k_u \subset N^b$.

Proof. First of all, we know that for *idp* to generate *uc*, there must be a processing step in ρ of the form (described in Lemma 4):

$$s_{\beta-1} \xrightarrow[idp \to \{(x:a_{idp}:resp_{\text{uc}})\}]{(a_{idp}:x:req_{\text{uc}}) \to idp} s_\beta \tag{D.8}$$

(for some addresses x, a_{idp} with $\beta < j$, where $resp_{\text{uc}}$ is an encrypted HTTP response with the body $\langle uc \rangle$). For the request req_{uc}, the method must be `POST` and the path component must be `/certreq`.

With Lemma 4 we know that req_{uc} was emitted by b, which is honest at this point in the run. With the same arguments as in (*) we can see that either *script_idp_pif* or the script *script_idp_ad* initiated req_{uc}.

For *script_idp_ad* it is easy to see that this script never sends a POST request to *idp*.

The script *script_idp_pif* can only send a POST request to `/certreq` in Line 47 of Algorithm D.8. In this case, the public key is chosen from the subterm `pubkeys` of the script's state. This subterm is only populated in Line 59 of Algorithm D.8. It can only be populated by a postMessage *pm* from an immediate parent window and from the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathsf{S} \rangle$ (given how a browser checks and transmits postMessages, see Line 94f. of Algorithm A.7). Further, the message in *pm* must be of the form $\langle n, \mathsf{pub}(k_u) \rangle$ where n is a nonce that was freshly chosen for a $\langle \texttt{genKeyPair}, n \rangle$ postMessage in Line 28 of Algorithm D.8.

Given that b's keymapping assigns the private key of LPO to the domain of LPO and with Lemma 3 we see that the only scripts that can send such a postMessage are *script_lpo_cif* and *script_lpo_ld*.

In the script *script_lpo_cif* (Algorithm D.4), postMessages of the form of *pm* can only be sent in Line 110 (the message sent in Line 105 would not carry the correct nonce for a response to a `genKeyPair` message).

The same holds true for the script *script_lpo_ld* (Algorithm D.4).

Therefore, the key k_u is a nonce that was chosen from the browser's nonces. □

Lemma 6. k_u does not leak from b.

Proof. As we have seen above, the key k_u was chosen either in the script *script_lpo_cif* or in the script *script_lpo_ld* running in the honest browser b.

In both scripts, any nonce that is chosen from the script's *nonces* will not be given to the script (as part of *nonces*) by the browser again, thus, the nonce was chosen freshly. Further, the nonce

is stored in the subterm `key` of the scriptstate and (besides the derivation of the public key) is only used to sign IAs.

There are no other scripts running in the origin of $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$. The (honest) browser b does not leak the scriptstate. Therefore, k_u does not leak from b. □

With Lemma 4, 5, and 6, we can see that only b knows k_u and the attacker cannot know k_u. Therefore, only b can create the $ia = \mathsf{sig}(\langle d_r, \mathtt{S} \rangle, k_u)$. As k_u is only accessible to scripts with the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$, only the script *script_lpo_cif* or the script *script_lpo_ld* can create $\mathsf{sig}(\langle d_r, \mathtt{S} \rangle, k_u)$. In both scripts, after creation, *ia* is sent in postMessage only to scripts that have the origin for which *ia* was created ($= \langle d_r, \mathtt{S} \rangle$). With Lemma 3 and the definition of relying parties (see Algorithm D.3) we see, that the only potential receiver is *script_rp_index*.

After receiving this `response` postMessage, *script_rp_index* stores the UC and the IA in the subterm called `cap` of its scriptstate (see Algorithm D.6, Line 48). After doing so, this subterm is read only in Line 73 (where only the identity is extracted) and in Line 85. There, the *ia* is sent to r (in the encrypted request req_{cap}).

The RP r, which is not corrupted, and the browser b do not leak *ia*. After receiving *ia*, r sends the newly created service token $\langle n, i \rangle$ to b, which ignores it (see Algorithm D.6 Line 91f.). Therefore, b and r do not leak $\langle n, i \rangle$.

Therefore, the attacker cannot know $\langle n, i \rangle$ in S_j, i.e., $\langle n, i \rangle \notin d_{N^{\mathsf{attacker}}}(S_j(\mathsf{attacker}))$. This is a contradiction to our assumption. ■

D.5. Proof of Theorem 2 (Security w.r.t. Session Integrity)

In order to prove Theorem 2, we have to proof that the BrowserID primary mode Web system satisfies security w.r.t. indirect session integrity (Definition 13).

We start from a processing step Q_{login} in which the browser is actually logged in and trace back on the causality chain (the session), i.e., for every processing step in the session, we analyze all other possible processing steps that can possibly lead to the later processing steps.

To start, we first need to assume that we have an arbitrary processing step in which the browser gets logged in: Let $\mathcal{BID}^s$ a BrowserID primary mode Web system, ρ an arbitrary run of $\mathcal{BID}^s$, Q_{login} an arbitrary processing step in ρ, b an arbitrary browser of $\mathcal{BID}^s$ that is honest in Q_{login}, r an arbitrary RP of $\mathcal{BID}^s$ that is honest in Q_{login}, and *id* an arbitrary identity.

To show the main implication of the session integrity property, we now assume that there exists some n such that $\mathsf{loggedIn}(Q_{\text{login}}, b, r, id, n)$. This means that b is currently processing a response (say, $resp_{\text{CAP}}$) to an XHR in a document under a secure origin of r that contains the script `script_rp_index`. We call the document RP-Doc. Further, b has received an HTTP(S)

response before that contains n in its body. This response must have a corresponding HTTP(S) request (say, req_{CAP}) that bears the XHR reference that is stored in the subterm *refXHRcap* in the scriptstate. This reference is only ever set in Lines 82ff. of Algorithm D.6. Hence, this request/response pair must use HTTPS.

We know that req_{CAP} is a POST request that carries some value *cap* in its body. The RP r only processes such a request in Lines 15ff. of Algorithm D.3. There, r checks whether *cap* is a valid CAP (the UC must be signed by the governor of the certified identity and the IA must be signed by the key certified in the UC) and extracts the identity id' for which the CAP was created. If the CAP is valid, an entry in the subterm *serviceTokens* is created such that $serviceTokens[n'] \equiv id'$. The value of n' is output inside an HTTPS response, which is $resp_{\text{CAP}}$. Hence, we have that $n \equiv n'$ and $id \equiv id'$ (as the subterm *serviceTokens* in the state of r is never changed, except by adding entries indexed by a fresh nonce).

Recall that the script `script_rp_index` is implemented as a state machine with the state stored in the subterm q of its state. The request req_{CAP} above is only ever created by this script in Lines 82ff. (of Algorithm D.6), which are only executed if the state machine is in state `sendCAP`. In these lines, the CAP is retrieved from the subterm *cap* in the scriptstate.

We now continue by analysing in which cases RP-Doc gets into the state `sendCAP` and how *cap* in the scriptstate is set. To reach the state `sendCAP`, there are only two possibilities:

Lines 77ff. In this case, RP-Doc must have been in state `loggedInUser`, which is only ever reached if Lines 71ff. have been processed in an earlier processing step. This state, in turn, can only be reached if the Lines 60ff. have been processed in an earlier processing step (say Q_{rpcap}). In these lines, also *cap* is stored in the state and there cannot be a different processing step after this processing step (up to Q_{login} in which this subterm is changed. In Q_{rpcap}, the value for *cap* is taken from a postMessage of the form $\langle \texttt{response}, cap \rangle$. This postMessage must have been sent by a document of LPO's origin.

LPO only ever delivers the scripts `script_lpo_cif` and `script_lpo_ld` out of which only `script_lpo_ld` ever creates such a postMessage. This postMessage is created in Line 107 of Algorithm D.5. We call the document in which this step is performed LD-Doc and the processing step in which this postMessage is created $Q_{\text{sendresponse}}$. Recall that the script `script_lpo_ld` is also implemented as a state machine with the state stored in the subterm q of the scriptstate. In LD-Doc, the postMessage above is only created if LD-Doc is in the state `createCAPforRP` and the value of the UC, which is used to create *cap* is taken from the subterm *uc* script's state. We know from above, that this UC must be a valid UC for the identity *id*. The only place where this subterm is actually set (it is initially $\bot$) is in Line 76 of Algorithm D.5. This line can only be reached if LD-Doc is in

state `runPIF`. We now trace back the state machine to this state. Recall that we are still analyzing the state `createCAPforRP` in our proof. There are only two places from which LD-Doc can transition to this state: Either Lines 76ff. (in state `runPIF`) or Lines 100ff. (in state `receiveLPOauthresponse`). The latter lines can only be reached if, in LD-Doc, Lines 92ff. (state `createCAPforLPO`) have been processed. These lines, however, can only be reached if Lines 76ff. (in state `runPIF`) have been processed. Hence, we know that the identity contained in the UC must match the subterm *email* of LD-Doc's state.

We now have to show that the subterm *email* only ever matches an identity owned by *b*:

Lemma 7. The value of s'.`email` in *script_lpo_ld* is always either one of the browser's identities or empty.

Proof. We show this by induction:

Base case: The value of s'.`email` is initially empty (see initial scriptstate).

Induction step: The value is set only in Lines 31 and 33. In the first case, the identity is chosen non-deterministically from the browser's identities *ids*, which are the identities that the browser owns (see Section D.3.5).

In the second case, the value of s'.`email` is taken from the localStorage, with the help of the key *idpnonce* that is taken from the sessionStorage. We can now show that what is retrieved from the localStorage is either empty or a previous value of s'.`email`:

First, we show that the value of *idpnonce*, taken from sessionStorage in Line 29, is always a nonce or empty: The browser's sessionStorage is separated by origins (and root windows), and therefore, only scripts under the origin of LPO have read or write access. Thus, the only two scripts that can possibly write the *idpnonce* value are *script_lpo_cif* and *script_lpo_ld*. The script *script_lpo_cif* does not write to sessionStorage. The script *script_lpo_ld* only writes to sessionStorage in Line 87. It only writes a fresh nonce (chosen in Line 85). Therefore, the value of *idpnonce* is always a nonce (or empty).

As we are already in the second case of the if-statement in Line 30 (we know that Line 33 was executed) *idpnonce* cannot be empty and must be a nonce.

Now, we can show that *localStorage*[*idpnonce*] is either empty or a previous value of s'.`email`: The browser's localStorage is separated by origins, and therefore, only scripts under the origin of LPO have read or write access. As above, the only two scripts that can write values to the localStorage are *script_lpo_cif* and *script_lpo_ld*. The script *script_lpo_cif* does not write to localStorage (it only removes subterms form localStorage in Line 47). We can thus focus on *script_lpo_ld*.

There are two lines where this script writes to the localStorage: Lines 112 and 86. We can safely ignore the first case, as it does not use a nonce as a key (but the fixed string `siteInfo` instead). In the latter case, it writes a value of $s'.\texttt{email}$.

This concludes the induction. □

As we know that the UC must have been issued for *id*, we know that the value of the subterm *email* cannot be empty. Hence, there must have been a processing step $Q_{\text{selectedID}}$ before (in ρ) for which we have that $\mathsf{selectedID}(Q_{\text{selectedID}}, b, \mathsf{governor}(id), id)$ and $\mathsf{selectedIdP}(Q_{\text{selectedID}}, b, \mathsf{governor}(id))$. In $Q_{\text{selectedID}}$, the script `script_lpo_ld` must have been in state `receiveContext`. Note that the document containing the script in $Q_{\text{selectedID}}$ is not necessarily LD-Doc (there exist possible traces in which the window containing LD-Doc is navigated). Hence, we call this document LD-Doc'.

We know that in LD-Doc', Lines 10ff. of Algorithm D.5 must have been executed in order to reach the state `receiveContext`. In these lines, this state can only be reached if a postMessage of the form $\langle \texttt{request}, \langle\rangle \rangle$ is processed from the inputs of LD-Doc'. When processing this postMessage, the value of the subterm *requestOrigin* of the state of LD-Doc' is set. This is the only place in which this subterm is modified (it is initially empty) and —as the state of LD-Doc' is changed to `receiveContext` when this postMessage is processed— there exist no possible trace in which this instance of LD-Doc' can ever reach the state `start` again. From above, we also know that, when LD-Doc' is triggered again, we are in processing step $Q_{\text{selectedID}}$. We further know that the CAP above was indeed issued for *r*. As the IA contained in the CAP is created for the origin that is stored in the subterm *requestOrigin* of LD-Doc and LD-Doc must also have been in state `start`. We further know that the value of the subterm *requestOrigin* must be the origin of the window that opened the window that contains LD-Doc (analogously LD-Doc'). Hence, LD-Doc and LD-Doc' are both top level windows (i.e., not contained in an iframe). Following the induction in the proof of Lemma 7 above, we know that LD-Doc' must have been in the same window tree as LD-Doc; otherwise the content of the sessionStorage is not preserved. We further know that LD-Doc must have received a postMessage of the form $\texttt{request}, \langle\rangle$ from its opener running a document under some secure origin of *r*. As we know that this must be `script_rp_index` in Line 57 of Algorithm D.6. We call the document this script is running in RP-Doc'. This script only sends this postMessage to a window it opened before when processing Lines 34ff. As we know that LD-Doc and LD-Doc' are located in the same top-level window, we can conclude that both, LD-Doc and LD-Doc' received these postMessages from the same RP-Doc'. Hence, the subterm *requestOrigin* is set to the same value in LD-Doc and LD-Doc'. We further know that before RP-Doc' creates

such postMessages, it must have selected the value `openLD` in Line 28 of Algorithm D.6 in some processing step Q_{start} before (in ρ).

Therefore, we have that $\exists o \in \mathsf{SSOSessions}(\rho, b, r)$ with $Q_{\text{start}} \in o$ and $Q_{\text{selectedID}} \in o$.

Lines 48ff. In this case, RP-Doc must have been in the state `default` and must have chosen to process a postMessage that was sent from a (secure) origin of LPO and from a window identified by the subterm *CIFindex* of the script's state. This subterm is initially $\perp$ and only ever set in Line 7ff., where it is set to the identifier of a freshly created iframe. We know that the postMessage is of the form $\langle \texttt{login}, cap' \rangle$. As above, there is only one script delivered by LPO that creates such a postMessage: `script_lpo_cif`. We call this document CIF-Doc here. In this script, such a postMessage is only created in Lines 123ff. of Algorithm D.4. Analogously to above, the script (again modelled as a state machine, similar to `script_lpo_ld`) must be in the state `createCAPforRP`. Analogously to above (as the structure of `script_lpo_cif` is *very* similar to `script_lpo_ld`, we can trace the UC contained in the CAP back to Lines 111ff. of Algorithm D.4 where the identity certified in the UC is compared to the subterm *email* in CIF-Doc's state. This value, however is only ever set once in this script (it is initially empty) in Line 64 where its value is read from localStorage under the value of the subterm *requestOrigin* of CIF-Doc's state. This subterm, again, is only ever set once in Line 20, where it is set to the origin of CIF-Doc's parent. As we know from above, that the parent is in fact RP-Doc, *requestOrigin* must have been set to a secure origin of r.

This part of the localStorage under the (secure) origin of LPO is only ever set in Line 112 of Algorithm D.5 (`script_lpo_ld`). Hence, we know that `script_lpo_ld` must have run exactly as outlined in the other case above (i.e., the user must have completed the login dialog to log in at r using the very same identity). Therefore, we have that $\exists o' \in \mathsf{SSOSessions}(\rho, b, r)$ and there exists a processing step $Q'_{\text{selectedID}} \in o'$ for which we have that $\mathsf{selectedID}(Q'_{\text{selectedID}}, b, \mathsf{governor}(id), id''')$, fulfilling our property for indirect session integrity.

■

E. Model and Analysis of BrowserID in Secondary Mode

E.1. Step-By-Step Description of BrowserID's Secondary Mode

We here provide a detailed description of the BrowserID flow including the LD and the CIF, but considering only the sIdP mode. As above, we leave out steps for fetching additional resources (like JavaScript files) and some less relevant postMessages.

As before, we assume that the user uses a "fresh" browser, i.e., the user has not been logged in before. Figure E.1 shows a sequence diagram of the run. In comparison to the high-level description with Phases (I) and (II) in Section 4.1, a new phase, the *Initialization*, is shown, which contains some initialization steps and some steps of the CIF for automatic creation of CAPs.

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

[1] Note that for postMessages the sender origin cannot be spoofed and is always correct (see [Hic15] for details).

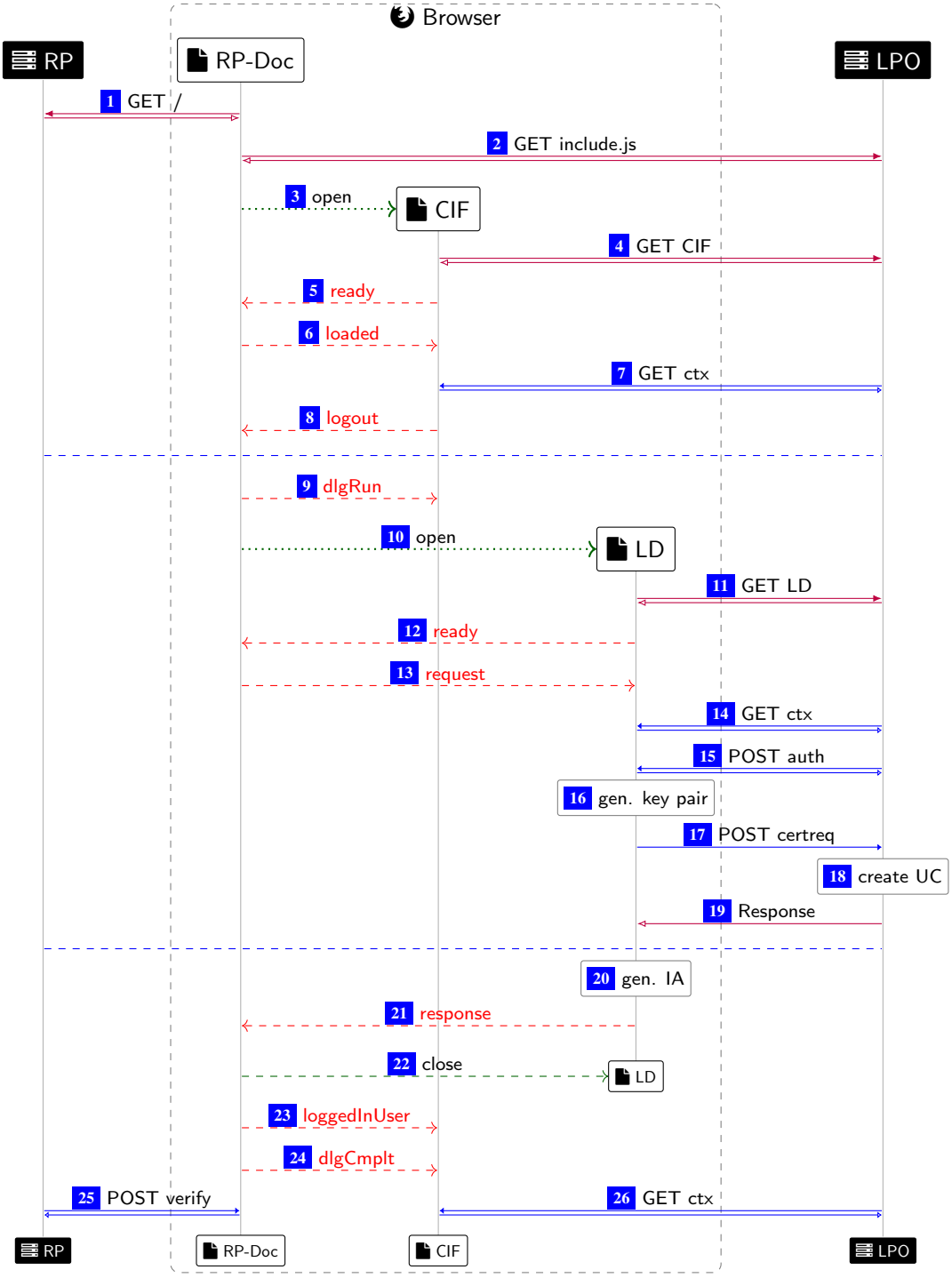


Figure E.1.: Detailed BrowserID implementation overview (secondary mode). (See Page 11 for notation.)

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

After this request, the dialog fetches the session context at LPO [14] (similar to what the CIF has done before). As the user is still not logged in, she is now asked to provide an email address and a password for LPO. These are then sent to LPO by an XHR [15]. If the credentials are accepted, a new key pair is generated by LD's JavaScript [16] and the public key along with the entered email address is sent to LPO [17] in order to get a UC. LPO generates a UC for the user [18] and sends this UC to LD [19] as a response to the request (see Section 4.1). Moreover, the key pair and the UC are stored in the localStorage under the origin of LPO.

Phase (II). Afterwards, an IA containing the so-called *audience* (the sender's origin of the `request` postMessage [13]) and some expiration date is created, signed (with the generated private key), and combined with the certificate to a CAP [20]. In the localStorage it is recorded that the user is logged in at RP with the current email address. The CAP and the email address are now sent back to RP-Doc in a `response` postMessage [21]. After this, the LD is closed [22].

The BrowserID JavaScript in RP-Doc informs the CIF that it now thinks that the email address received in the `response` postMessage is logged in [23]. Next, it tells the CIF that the LD is now closed [24], by which the CIF is awoken from pausing. The CIF then fetches the session context again [26] (as in [7]) in order to perform some additional checks (see below).

It is not specified in the BrowserID system how the RP-Doc has to process the CAP received in step [21]. Typically, as already mentioned in Section 4.1, the RP-Doc would send the CAP to the RP's server [25], which then can verify the CAP. If successful, RP can consider the user (with the email address mentioned in the CAP) to be logged in and send her some token, the RP service token (as introduced in Section 4.1).

E.1.1. Additional Checks

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

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

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

The LD sends [12] only to its opener (RP-Doc) without providing any receiver's origin to check. After this, the LD accepts only one `request` postMessage [13] and blocks any further incoming postMessages. The sender's origin of the `request` postMessage [13] is used by LD to determine the receiver's origin of the `response` postMessage [21]. LD also fixes the receiver of [21] to be its opener. When RP-Doc sends the `request` postMessage [13] to the LD, it sets the receiver's origin to be LPO in the postMessage. However, any postMessage RP-Doc expects to be sent by LD is not checked (see also Section 4.5.1).

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

We also note that step [24] triggers two checks in the CIF: First, the CIF checks the current login status at LPO, by fetching the session context [26]. Second, the CIF compares the email address received in [23] to the one that is marked as being logged in at RP in the localStorage (under the origin of LPO). If in one of the checks the user is considered to be not logged in, a `logout` postMessage is sent to RP-Doc (similarly to [8]). Otherwise, if in the second check a mismatch is detected, the CIF creates a new CAP according to the information in the localStorage and sends it as a so-called `login` postMessage to RP-Doc. Whether this CAP is used by RP-Doc or the one received in step [21] depends on the way the RP-Doc uses the API provided by BrowserID. One possibility (which is considered in the BrowserID test implementation) is that RP-Doc relays all received CAPs to the RP server with an XHR. The RP server, as already mentioned above, would then verify each CAP it receives and issue an RP service token every time. This is also what is done in our model of the BrowserID system.

E.1.2. Automatic CAP Creation

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

E.1.3. LPO Session

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

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

E.1.4. Logout

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

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

E.2. Model of BrowserID in Secondary Mode

In this section, we provide the full BrowserID model for the secondary mode, i.e., the SSO Web system $\mathcal{BID}^s = (\mathcal{W}^s, \mathcal{S}, \mathsf{script}, E_0)$, and its security properties. Again, we note that our model considers the BrowserID system with the fixes proposed in Section 4.5.1.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_rp_index	`script_rp_index`
script_lpo_cif	`script_lpo_cif`
script_lpo_ld	`script_lpo_ld`

Table E.1.: List of scripts in $\mathcal{S}$ and their respective string representations for the secondary mode of BrowserID.

Recall that the system $\mathcal{W}^p = \mathsf{Hon} \dot\cup \mathsf{Web} \dot\cup \mathsf{Net}$ contains the Web server for LPO, a finite set B of Web browsers, and a finite set RP of Web servers for the relying parties, with

$$\mathsf{Hon} := \mathsf{B} \dot\cup \mathsf{RP} \dot\cup \underbrace{\{\mathsf{LPO}\}}_{=:\mathsf{IDP}} \dot\cup \underbrace{\mathsf{DNS}}_{=\emptyset} \dot\cup \underbrace{\mathsf{Other}}_{=\emptyset} .$$

For the analysis of authentication and session integrity, we consider a BrowserID secondary mode Web system with one network attacker, i.e., $\mathsf{Web} := \emptyset$, and $\mathsf{Net} := \{\mathtt{attacker}\}$. DNS servers are assumed to be dishonest, and hence, are subsumed by attacker.

Table E.1 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 A.8.

E.2.1. Addresses and Domain Names

Similar to the model of BrowserID in primary mode, the set IPs contains for LPO, attacker, every relying party in RP, and every browser in B one address each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains one domain for LPO, one for every relying party in RP, and a finite set of domains for attacker. Browsers (in B) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively. If dom or addr returns a set with only one element, we often write $\mathsf{dom}(x)$ or $\mathsf{addr}(x)$ to refer to the element.

E.2.2. Keys and Secrets

Let $\mathsf{tlskey} \colon \mathsf{Doms} \to K_{\text{private}}$ be an injective mapping that assigns a private key to every domain. Atomic processes are given the private keys for their domain in the following form: For an atomic DY process p we define

$$\mathit{tlskeys}^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle .$$

The set $\mathsf{Secrets} \subseteq \mathcal{N}$ is the set of passwords (secrets) the browsers share with LPO. Let ID be the finite set of a email address (IDs) of the form $\langle name, d \rangle$, with $name \in \mathbb{S}$ and $d \in \mathsf{Doms}$, registered at LPO. As mentioned in Section 4.4.2, different browsers own disjoint sets of secrets and different secrets are assigned disjoint sets of identities. Let $\mathsf{ownerOfSecret} : \mathsf{Secrets} \to \mathsf{B}$ denote the mapping that assigns to each secret the browser that owns this secret. Let $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Secrets}$ denote the mapping that assigns to each identity the secret it belongs to. Now, we define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}, i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity.

As in the primary mode, the attacker initially does not own a secret, but can corrupt any number of browsers in a run, and hence, get hold of secrets as well.

E.2.3. Corruption

RPs 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 is *honest* if the according part of their state ($s.\mathtt{corrupt}$) is $\bot$, and that they are corrupted otherwise.

Recall that browsers can, as explained in Section 2.5, become corrupted as well.

E.2.4. Attackers

Attackers are modeled similar as in the primary mode as explained in Appendix D.3.4. In the secondary mode of BrowserID, we consider only a network attacker as we in this mode do not analyze privacy as discussed in Section 4.4.2.

We allow the (network) attacker to listen to/spoof all available IP addresses, and hence, define $I^{\mathsf{attacker}} = \mathsf{IPs}$. His initial state is $s_0^{\mathsf{attacker}} = \langle attdoms, pubkeys \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker and *pubkeys* is a sequence of all domains and the corresponding public keys. All other parties use the attacker as a DNS server.

E.2.5. Browsers

Each $b \in \mathsf{B}$ is a Web browser as defined in Appendix A.6, with $I^b := \{\mathsf{addr}(b)\}$ being its address and the initial state s_0^b defined as follows: the key mapping maps every domain to its public key, according to the mapping tlskey; the DNS address is $\mathsf{addr}(\mathtt{attacker})$; the secrets are those owned by b (as defined above) and they are indexed by the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$; *sts* is $\langle \mathsf{dom}(\mathsf{LPO}) \rangle$. (Without the latter, the attacker could trivially inject, by an HTTP response, its own `browserid_state` cookie and by this violate the session integrity property.)

E.2.6. LPO

LPO is a an atomic DY process $(I^{\mathsf{LPO}}, Z^{\mathsf{LPO}}, R^{\mathsf{LPO}}, s_0^{\mathsf{LPO}})$ with the address $I^{\mathsf{LPO}} = \{\mathsf{addr}(\mathsf{LPO})\}$. The initial state s_0^{LPO} of LPO contains the private key of its domain, its signing key k^{LPO}, all secrets in Secrets and the corresponding sequences of IDs. (LPO does not need the public keys of other parties, which is why we omit them from LPO's initial state.). HTTP responses by LPO can contain, besides complex terms (e.g., for XHR responses), strings representing scripts, namely the script `script_LPO_cif` run in the CIF and the script `script_LPO_ld` run in the LD. These scripts are defined in Appendices E.2.8 and E.2.8, respectively.

LPO maintains client sessions in the same way as LPO in the primary mode of BrowserID as explained in Section 4.4.1.

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

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

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

`GET /ctx`. LPO replies with a session context. More precisely, LPO first checks if a cookie `browserid_state` was sent within this request and if its value identifies a session within LPO's state. If such a session exists, LPO responds to such a request with the list of (authenticated) identities for this session,[2] the xsrfToken, and a `Set-Cookie` header, which sets the `browserid_state` cookie. If no cookie `browserid_state` was sent in the request, or if the value of the cookie `browserid_state` does not identify a session within LPO's state, LPO first creates a new session. Such a new session contains a fresh nonce as a session identifier, the empty sequence $\langle\rangle$ of identities, and a fresh nonce as a xsrfToken. Once such a session is created, LPO responds as above.

`POST /auth`. This request is sent to authenticate a session at LPO. A request to this interface has to contain some secret $sec \in \mathsf{Secrets}$ in its body. The request also has to contain the cookie `browserid_state` which has to refer to some session in the state of LPO. Moreover, the request has to contain an xsrfToken in its body which has to match the one recorded in the considered session in LPO's state. The session recorded in the state of LPO is then modified to include the sequence of all identities associated with *sec*. The response to such a request contains some static acknowledgment.

[2] In the real implementation, the session context only contains a flag indicating the authentication state of the session. However, another GET request interface is available to retrieve the list of authenticated identities for the current session. Here, for simplicity, we right away provide all authenticated identities in the session context.

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

We now define LPO formally as an atomic DY process $(I^{\mathsf{LPO}}, Z^{\mathsf{LPO}}, R^{\mathsf{LPO}}, s_0^{\mathsf{LPO}})$. As already mentioned, we define $I^{\mathsf{LPO}} = \{\mathsf{addr}(\mathsf{LPO})\}$.

In order to define the set Z^{LPO} of states of LPO, we repeat the definition of an LPO session context from the primary mode of BrowserID (see Definition 75).

Definition 87. A term of the form $\langle ids, xsrfToken \rangle$ with $ids \subset^{\langle\rangle} \mathsf{ID}$ and $xsrfToken \in \mathcal{N}$ is called an *LPO session context*. We denote the set of all LPO session contexts by $\mathsf{LPOSessionCTXs}$. ◇

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

Definition 88. A state $s \in Z^{\mathsf{LPO}}$ of LPO is a term of the form $\langle tlskey, signkey, sessions, secrets \rangle$ where $tlskey = \mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO}))$, $signkey = k^{\mathsf{LPO}}$, $sessions \in [\mathcal{N} \times \mathsf{LPOSessionCTXs}]$, and $secrets \in [\mathsf{Secrets} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary which assigns to every secret $sec \in \mathsf{Secrets}$ the sequence of all identities associated with sec.[3]

The *initial state* s_0^{LPO} *of LPO* is a state of LPO with $s_0^{\mathsf{LPO}}.\mathtt{sessions} = \langle\rangle$. ◇

We specify the relation $R^{\mathsf{LPO}} \subseteq (\mathcal{E} \times Z^{\mathsf{LPO}}) \times (2^{\mathcal{E}} \times Z^{\mathsf{LPO}})$ of LPO in Algorithm E.1.

E.2.7. Relying Parties

A relying party $r \in \mathsf{RP}$ is a Web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the address $I^r := \{\mathsf{addr}(r)\}$. Its initial state s_0^r contains its domain, the private key associated with its domain, the public key of LPO, and the set of service token it has provided. The definition of R^r again follows the description in Appendix E.1. RP only accepts messages sent over HTTPS. Whenever r receives a `GET` message, it returns the script `script_RP_index` (see below) and sets the `Strict-Transport-Security` header. If r receives an HTTPS `POST` message, it checks if

[3] This is a simplified variant of the "user database" used for IdPs in the primary mode of BrowserID.

Algorithm E.1: Relation of LPO R^{LPO}.

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

let $s' := s$
if $m \equiv \texttt{TRIGGER}$ **then** → Triggers a (non-deterministic) logout or session expiration
  if $s'.\texttt{sessions} \not\equiv \langle\rangle$ **then**
    let *sessionid* $\leftarrow \{id_j | id_j \in s'.\texttt{sessions}\}$
    if *sessionid* $\in^{\langle\rangle} s'.\texttt{sessions}$ **then**
      let *choice* $\leftarrow \{\texttt{logout}, \texttt{expire}\}$
      if *choice* $\equiv \texttt{logout}$ **then**
        let *session* $:= s'.\texttt{sessions}[sessionid]$
        let $session[ids] := \langle\rangle$
        let $s'.\texttt{sessions}[sessionid] := session$
        stop $\{\}, s'$
      else
        remove the element with key *sessionid* from the dictionary $s'.\texttt{sessions}$
        stop $\{\}, s'$
      end if
    end if
  end if
else
  let m_{dec}, k' **such that** $\langle m_{\text{dec}}, k'\rangle \equiv \mathsf{dec_a}(m, s'.\texttt{tlskey})$ **if possible; otherwise stop** $\{\}, s$
  let *n*, *method*, *path*, *params*, *headers*, *body* **such that** $\langle \texttt{HTTPReq}, n, method, \mathsf{dom}(\mathsf{LPO}), path, params, headers, body\rangle \equiv m_{\text{dec}}$ **if possible; otherwise stop** $\{\}, s$
  if $method \equiv \texttt{GET} \wedge path \equiv /\texttt{cif}$ **then** → Deliver CIF script
    let $m' :=$ $\mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\langle \texttt{Strict-Transport-Security}, \top\rangle\rangle, \langle \texttt{script\_LPO\_cif}, initState_{cif}\rangle\rangle, k')$ where $initState_{cif}$ is the initial scriptstate of *script_LPO_cif* according to Definition 90.
    let $E := \{anf, a, m'\}$
    stop E, s'
  else if $method \equiv \texttt{GET} \wedge path \equiv /\texttt{ld}$ **then** → Deliver LD script.
    let $m' :=$ $\mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\langle \texttt{Strict-Transport-Security}, \top\rangle\rangle, \langle \texttt{script\_LPO\_ld}, initState_{ld}\rangle\rangle, k')$ where $initState_{ld}$ is the initial scriptstate of *script_LPO_ld* according to Definition 91.
    let $E := \{\langle f, a, m'\rangle\}$
    stop E, s'
  else if $method \equiv \texttt{GET} \wedge path \equiv /\texttt{ctx}$ **then** → Deliver context information.
    let $sessionid := headers[\texttt{Cookie}][\texttt{browserid\_state}]$
    if $sessionid \notin^{\langle\rangle} s'.\texttt{sessions}$ **then** → Create new session if needed.
      let $sessionid := \nu_1$
      let $ids := \langle\rangle$
      let $xsrfToken := \nu_2$
      let $s'.\texttt{sessions} := s'.\texttt{sessions} +^{\langle\rangle} \langle sessionid, \langle ids, xsrfToken\rangle\rangle$
    end if
    let $result := s'.\texttt{sessions}[sessionid]$
    let $headers' := \langle\langle \texttt{Strict-Transport-Security}, \top\rangle, \langle \texttt{Set-Cookie}, \langle\langle \texttt{browserid\_state}, \langle sessionid, \top, \top, \top\rangle\rangle\rangle\rangle\rangle$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, headers', result\rangle, k')$
    let $E := \{\langle f, a, m'\rangle\}$
    stop E, s'

This algorithm is continued on the next page.

```
42:     else if method ≡ POST ∧ path ≡ /auth then    → Authenticate session.
43:         let sessionid := headers[Cookie][browserid_state]
44:         let secret, xsrfToken such that ⟨secret, xsrfToken⟩ ≡ body if possible; otherwise stop {}, s
45:         if sessionid ∈^⟨⟩ s'.sessions then
46:             if secret ∈^⟨⟩ s'.secrets ∧ s'.sessions[sessionid].xsrfToken ≡ xsrfToken then
47:                 let ids := s'.secrets[secret]
48:                 let s'.sessions[sessionid].ids := ids
49:                 let m' := enc_s(⟨HTTPResp, n, 200, ⟨⟨Strict-Transport-Security, ⊤⟩⟩, ⊤⟩, k')
50:                 let E := {⟨f, a, m'⟩}
51:                 stop E, s'
52:             end if
53:         end if
54:     else if method ≡ POST ∧ path ≡ /certreq then    → Sign pubkey, deliver UC
55:         let sessionid := headers[Cookie][browserid_state]
56:         let ids := s'.sessions[sessionid].ids
57:         let xsrfToken := s'.sessions[sessionid].xsrfToken
58:         let id, pubkey, xsrfToken' such that ⟨id, pubkey, xsrfToken'⟩ ≡ body if possible;
              ↪ otherwise stop {}, s
59:         if id ∈^⟨⟩ ids ∧ xsrfToken ≡ xsrfToken' then
60:             let uc := sig(⟨id, pubkey⟩, s'.signkey)
61:             let m' := enc_s(⟨HTTPResp, n, 200, ⟨⟨Strict-Transport-Security, ⊤⟩⟩, uc⟩, k')
62:             let E := {⟨f, a, m'⟩)}
63:             stop E, s'
64:         end if
65:     end if
66: end if
67: stop {}, s
```

(1) the message contains a valid CAP for r, and (2) the header of the message contains an `Origin` header which only contains a single origin and this origin is r's domain with HTTPS. If this check is successful, r responds with a token of the form $\langle n, i\rangle$ (sent in the body of the response), where $i \in \mathsf{ID}$ is the identity for which the CAP was issued and n is a freshly chosen nonce. We call, as mentioned in Section 4.4.2, $\langle n, i\rangle$ an *RP service session (for id i).* As mentioned, r keeps a list of such tokens in its state. Intuitively, a client in possession of such a token can use the service of r for identity i (e.g., access data of i at r).

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

Definition 89. A state $s \in Z^r$ of an RP r is a term of the form

$$\langle tlskey, domain, pubkLPO, serviceTokens\rangle$$

Algorithm E.2: Relation of an RP R^r.

Input: $\langle a, f, m\rangle, s$
1: **let** $s' := s$
2: **let** m_{dec}, k' **such that** $\langle m_{\text{dec}}, k'\rangle \equiv \mathsf{dec_a}(m, s'.\texttt{tlskey})$ **if possible; otherwise stop** $\{\}, s$
3: **let** n, *method*, *path*, *params*, *headers*, *body* **such that** $\langle \texttt{HTTPReq}, n, method, \texttt{s}'.\texttt{domain}, path, params, headers, body\rangle \equiv m_{\text{dec}}$ **if possible; otherwise stop** $\{\}, s$
4: **if** $method \equiv \texttt{GET}$ **then** → Deliver RP's index script
5: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\langle \texttt{Strict-Transport-Security}, \top\rangle\rangle, \langle \texttt{script\_RP\_index}, initState_{rp\_index}\rangle\rangle, k')$ where $initState_{rp\_index}$ is the initial scriptstate of *script_RP_index* according to Definition 92.
6: **let** $E := \{\langle f, a, m'\rangle\}$
7: **stop** E, s'
8: **else if** $(method \equiv \texttt{POST}) \wedge (headers \equiv \langle\langle \texttt{Origin}, \langle s'.\texttt{domain}, \texttt{S}\rangle\rangle\rangle)$ **then** → Check received CAP
9: **let** *uc*, *ia* **such that** $\langle uc, ia\rangle \equiv body$ **if possible; otherwise stop** $\{\}, s$ → Extract UC and IA
10: **let** $i := \pi_1(\mathsf{extractmsg}(uc))$ → Extract id from UC
11: **let** $pku := \pi_2(\mathsf{extractmsg}(uc))$ → Extract pubkey from UC
12: **let** $o := \mathsf{extractmsg}(ia)$ → Extract audience from IA
13: **if** $(\mathsf{checksig}(uc, s'.\texttt{pubkLPO}) \equiv \top) \wedge (\mathsf{checksig}(ia, pku) \equiv \top \wedge o \equiv \langle s'.\texttt{domain}, \texttt{S}\rangle)$ **then**
14: **let** $n' := \nu_1$ → Issue service token
15: **let** $s'.\texttt{serviceTokens} := s'.\texttt{serviceTokens} + \langle n', i\rangle$
16: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{n}', \texttt{i}\rangle\rangle, k')$
17: **let** $E := \{\langle f, a, m'\rangle\}$
18: **stop** E, s'
19: **end if**
20: **end if**
21: **stop** $\{\}, s$

where $tlskey = \mathsf{tlskey}(\mathsf{dom}(r))$, $domain = \mathsf{dom}(r)$, $pubkLPO = \mathsf{pub}(\mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO})))$, and $serviceTokens \in [\mathcal{N} \times \mathbb{S}]$.

The *initial state* s_0^r *of* r is a state of r with $s_0^r.\texttt{serviceTokens} = \langle\rangle$. ◇

As for the primary mode, we define the function serviceSessions that extracts service sessions from an RP's state as the projection that extracts the subterm *serviceTokens*.

We specify the relation $R^r \subseteq (\mathcal{E} \times Z^r) \times (2^{\mathcal{E}} \times Z^r)$ of r in Algorithm E.2.

E.2.8. BrowserID Scripts

As already mentioned in Section 4.4.2, the set $\mathcal{S}$ of the Web system $\mathcal{BID}^s = (\mathcal{W}^s, \mathcal{S}, \mathsf{script}, E_0)$ consists of the scripts R^{att}, *script_RP_index*, *script_LPO_cif*, and *script_LPO_ld* with string representations `att_script`, `script_RP_index`, `script_LPO_cif`, and `script_LPO_ld` (defined by script, see also Table E.1). The script R^{att} is the attacker script (see Section 2.4). The formal model of the latter two scripts follows the description in Appendix E.1 in a straightforward way. The script *script_RP_index* defines the script of the RP index page. In reality, this

page has its own script(s) and includes a script from LPO. In our model, we combine both scripts to *script_RP_index*. In particular, this script is responsible for creating the CIF and the LD iframes/subwindows, whose content (scripts) are loaded from LPO.

In what follows, the scripts *script_RP_index*, *script_LPO_cif*, and *script_LPO_ld* are defined formally. We note that the scripts of LPO also make use of localStorage as described below.

LocalStorage Under the Origin of LPO. The localStorage under the origin of LPO used by the scripts *script_LPO_cif* and *script_LPO_ld* is organized in similar, but simplified way as in the primary mode of BrowserID: the localStorage is a dictionary containing only one entry. This entry consists of the key `siteInfo` and (as its value) a dictionary where this dictionary has origins as keys with identities as values indicating that a certain identity (of the user) is logged in at the referenced origin. Here is an example a possible localStorage.

Example 9.

$$\langle\langle \texttt{siteInfo}, \langle\langle\langle \texttt{domain}_{\texttt{RP1}}, \texttt{S}\rangle, \texttt{id}_1\rangle, \langle\langle \texttt{domain}_{\texttt{RP2}}, \texttt{S}\rangle, \texttt{id}_1\rangle, \langle\langle \texttt{domain}_{\texttt{RP3}}, \texttt{S}\rangle, \texttt{id}_2\rangle\rangle\rangle\rangle \quad \text{(E.1)}$$

This example shows a localStorage under the origin of LPO, indicating that the user is logged in at $\texttt{domain}_{\texttt{RP1}}$ and $\texttt{domain}_{\texttt{RP2}}$ with $\texttt{id}_1$ and at $\texttt{domain}_{\texttt{RP3}}$ with $\texttt{id}_2$ (using HTTPS).

login.persona.org Communication Iframe Script (script_LPO_cif). This script models the script run in the CIF, as sketched in Appendix E.1.

We first describe the structure of the internal scriptstate of the script *script_LPO_cif*.

Definition 90. A scriptstate s of *script_LPO_cif* is a term of the form

$$\langle q, \mathit{parentOrigin}, \mathit{loggedInUser}, \mathit{pause}, \mathit{context}, \mathit{key}, \mathit{handledInputs}, \mathit{refXHRctx}, \mathit{refXHRcert}\rangle$$

where $q \in \mathbb{S}$, $\mathit{parentOrigin} \in \mathsf{Origins} \cup \{\bot\}$, $\mathit{loggedInUser} \in \mathsf{ID} \cup \{\langle\rangle, \bot\}$, $\mathit{pause} \in \{\top, \bot\}$, $\mathit{context} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{key} \in \mathcal{N} \cup \{\bot\}$, $\mathit{handledInputs} \subset^{\langle\rangle} \mathbb{N}$, $\mathit{refXHRctx}, \mathit{refXHRcert} \in \mathcal{N} \cup \{\bot\}$.

The initial state $\mathit{initState}_{\mathit{cif}}$ of *script_LPO_cif* is the state $\langle \texttt{init}, \bot, \bot, \bot, \bot, \bot, \langle\rangle, \bot, \bot\rangle$. $\diamond$

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

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

$q =$ `default` is the state to which `script_LPO_cif` always returns to. This state handles all postMessages the CIF expects to receive. If the postMessage received was sent from the parent window of the CIF, it behaves as follows:

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

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

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

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

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

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

$q =$ `receiveContext` expects an XHR response as input containing the session context. This context is saved as the current context in the scriptstate. The script transitions to the next state `checkAndEmit`.

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

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

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

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

We now specify the relation *script_LPO_cif* $\subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the CIF's scripting process formally.

Just like all scripts, as explained in Section 2.5 (see also Algorithm A.7 for the formal specification), the input term this script obtains from the browser contains the cleaned tree of the browser's windows and documents *tree*, the nonce of the current document *docnonce*, its own scriptstate *scriptstate* (as defined in Definition 90), a sequence of all inputs *scriptinput* (also containing already handled inputs), a dictionary *cookies* of all accessible cookies of the document's domain, the localStorage *localStorage* belonging to the document's origin, and the secrets *secrets* of the document's origin. The script returns a new scriptstate s', a new set of cookies *cookies*$'$, a new localStorage *localStorage*$'$, and a term *command* denoting a command to the browser.

The relation of the script is defined in Algorithm E.3.

login.persona.org Login Dialog Script (script_LPO_ld).

The script *script_LPO_ld* models the script that runs in the LD. Its formal specification, presented next, follows the one presented above for *script_LPO_cif*.

Definition 91. A *scriptstate s of script_LPO_ld* is a term of the form $\langle q, \mathit{requestOrigin}, \mathit{context}, \mathit{key}, \mathit{handledInputs}, \mathit{refXHRctx}, \mathit{refXHRauth}, \mathit{refXHRcert}\rangle$ with $q \in \mathbb{S}$, $\mathit{requestOrigin} \in \mathsf{Origins} \cup \{\bot\}$, $\mathit{context} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{key} \in \mathcal{N} \cup \{\bot\}$, $\mathit{handledInputs} \subset^{\langle\rangle} \mathbb{N}$, $\mathit{refXHRctx}, \mathit{refXHRauth}, \mathit{refXHRcert} \in \mathcal{N} \cup \{\bot\}$.

The initial state $\mathit{initState}_{ld}$ is the state $\langle \texttt{init}, \bot, \bot, \bot, \langle\rangle, \bot, \bot, \bot\rangle$. ◇

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

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

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

Algorithm E.3: Relation of *script_LPO_cif*.

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

let $s' := scriptstate$
let $cookies' := cookies$
let $localStorage' := localStorage$
switch $s'.\texttt{q}$ **do**
  case `init`
    let $command := \langle \texttt{POSTMESSAGE}, \mathsf{PARENTWINDOW}(tree, docnonce), \langle \texttt{cifready}, \langle\rangle\rangle, \bot\rangle$
    let $s'.\texttt{q} := \texttt{default}$
    stop $\langle s', cookies', localStorage', sessionStorage, command\rangle$
  case `default`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $\pi_1(input) \equiv \texttt{POSTMESSAGE}$ **then**
      let $senderWindow := \pi_2(input)$
      let $senderOrigin := \pi_3(input)$
      let $m := \pi_4(input)$
      if $senderWindow \equiv \mathsf{PARENTWINDOW}(tree, docnonce)$ **then**
        switch m **do**
          case $\langle \texttt{loaded}, id\rangle$
            let $s'.\texttt{parentOrigin} := senderOrigin$
            let $s'.\texttt{loggedInUser} := id$
            if $s'.\texttt{pause} \equiv \top$ **then**
              stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
            else if $s'.\texttt{context} \equiv \bot$ **then**
              let $s'.\texttt{q} := \texttt{fetchContext}$
              stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
            else
              let $s'.\texttt{q} := \texttt{checkAndEmit}$
              stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
            end if
          case $\langle \texttt{dlgRun}, \langle\rangle\rangle$
            let $s'.\texttt{pause} := \top$
            stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
          case $\langle \texttt{dlgCmplt}, \langle\rangle\rangle$
            let $s'.\texttt{pause} := \bot$
            let $s'.\texttt{q} := \texttt{fetchContext}$
            stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
          case $\langle \texttt{loggedInUser}, id\rangle$
            let $s'.\texttt{loggedInUser} := id$
            stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
          case $\langle \texttt{logout}, \langle\rangle\rangle$
        end if
      end if
  case `fetchContext`
    let $s'.\texttt{refXHRctx} := \lambda_1$
    let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{\mathsf{LPO}}_{/\texttt{ctx}}, \texttt{GET}, \langle\rangle, s'.\texttt{refXHRctx}\rangle$
    let $s'.\texttt{q} := \texttt{receiveContext}$
    stop $\langle s', cookies', localStorage', sessionStorage, command\rangle$

This algorithm is continued on the next page.

47: **case** `receiveContext`
48: **let** $input, s' :=$ CHOOSEINPUT(s', *scriptinputs*)
49: **if** $(\pi_1(input) \equiv \texttt{XMLHTTPREQUEST}) \wedge (\pi_3(input) \equiv s'.\texttt{refXHRctx})$ **then**
50: **let** $s'.\texttt{context} := \pi_2(input)$
51: **let** $s'.\texttt{q} :=$ `checkAndEmit`
52: **stop** $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
53: **end if**
54: **case** `checkAndEmit`
55: **let** $lid :=$ *localStorage*$'[\texttt{siteInfo}][s'.\texttt{parentOrigin}]$
56: **if** $(lid \not\equiv \langle\rangle) \wedge (s'.\texttt{loggedInUser} \notin \{\langle\rangle, \bot\} \Rightarrow (s'.\texttt{loggedInUser} \not\equiv lid)) \wedge (\pi_1(s'.\texttt{context}) \not\equiv \langle\rangle)$ **then**
57: **let** $s'.\texttt{q} :=$ `requestUC`
58: **stop** $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
59: **else if** $s'.\texttt{loggedInUser} \equiv \langle\rangle$ **then**
60: **let** $s'.\texttt{q} :=$ `default`
61: **stop** $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
62: **else**
63: **let** $s'.\texttt{q} :=$ `sendLogout`
64: **stop** $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
65: **end if**
66: **case** `requestUC`
67: **let** $id :=$ *localStorage*$'[\texttt{siteInfo}][s'.\texttt{parentOrigin}]$
68: **let** $s'.\texttt{key} := \lambda_2$
69: **let** $body := \langle id, \text{pub}(s'.\texttt{key}), s'.\texttt{context.xsrfToken}\rangle$
70: **let** $s'.\texttt{refXHRcert} := \lambda_3$
71: **let** $command := \langle \texttt{XMLHTTPREQUEST}, \text{URL}^{\text{LPO}}_{/\texttt{certreq}}, \texttt{POST}, body, s'.\texttt{refXHRcert}\rangle$
72: **let** $s'.\texttt{q} :=$ `receiveUC`
73: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$
74: **case** `receiveUC`
75: **let** $input, s' :=$ CHOOSEINPUT(s', *scriptinputs*)
76: **if** $(\pi_1(input) \equiv \texttt{XMLHTTPREQUEST}) \wedge (\pi_3(input) \equiv s'.\texttt{refXHRcert})$ **then**
77: **let** $uc := \pi_2(input)$
78: **let** $ia :=$ sig$(s'.\texttt{parentOrigin}, s'.\texttt{key})$
79: **let** $cap := \langle uc, ia\rangle$
80: **let** *command* :=
$\langle \texttt{POSTMESSAGE}, \text{PARENTWINDOW}(tree, docnonce), \langle \texttt{login}, cap\rangle, s'.\texttt{parentOrigin}\rangle$
81: **let** $s'.\texttt{q} :=$ `default`
82: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$
83: **end if**
84: **case** `sendLogout`
85: **let** $command := \langle \texttt{POSTMESSAGE}, \text{PARENTWINDOW}(tree, docnonce), \langle \texttt{logout}, \langle\rangle\rangle, \bot\rangle$
86: **let** $s'.\texttt{q} :=$ `default`
87: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$
88: **stop** $\langle scriptstate, cookies, localStorage, sessionStorage, \langle\rangle\rangle$

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

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

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

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

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

$q \equiv$ `null` do nothing.

We formally specify the relation $\mathit{script\_LPO\_ld} \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the LD's scripting process in Algorithm E.4.

Relying Party Web Page Script (script_RP_index).

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

Definition 92. A scriptstate s of *script_RP_index* is a term of the form

$$\langle q, \mathit{CIFindex}, \mathit{LDindex}, \mathit{dialogRunning}, \mathit{cap}, \mathit{handledInputs}, \mathit{refXHRcap}\rangle$$

with $q \in \mathbb{S}$, $\mathit{CIFindex} \in \mathbb{N} \cup \{\bot\}$, $\mathit{dialogRunning} \in \{\top, \bot\}$, $\mathit{cap} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{handledInputs} \subset^{\langle\rangle} \mathbb{N}$, $\mathit{refXHRcap} \in \mathcal{N} \cup \{\bot\}$.

We call s the initial scriptstate of *script_RP_index* iff $s \equiv \langle \texttt{init}, \bot, \bot, \bot, \langle\rangle, \langle\rangle, \bot\rangle$. $\diamond$

Before we provide the formal specification of the relation that defines the behavior of the script *script_RP_index*, we present an informal description. As above, the behavior mainly depends on the state q the script is in.

Algorithm E.4: Relation of *script_LPO_ld*.

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

let $s' := scriptstate$
let $cookies' := cookies$
let $localStorage' := localStorage$
switch $s'.\mathtt{q}$ **do**
  case `init`
    let $command := \langle \mathtt{POSTMESSAGE}, \mathsf{OPENERWINDOW}(tree, docnonce), \langle \mathtt{ldready}, \langle\rangle\rangle, \bot\rangle$
    let $s'.\mathtt{q} := \mathtt{start}$
    stop $\langle s', cookies', localStorage', sessionStorage, command\rangle$
  case `start`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $\pi_1(input) \equiv \mathtt{POSTMESSAGE}$ **then**
      let $senderWindow := \pi_2(input)$
      let $senderOrigin := \pi_3(input)$
      let $m := \pi_4(input)$
      if $m \equiv \langle \mathtt{request}, \langle\rangle\rangle$ **then**
        let $s'.\mathtt{requestOrigin} := senderOrigin$
        let $s'.\mathtt{refXHRctx} := \lambda_1$
        let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{ctx}}, \mathtt{GET}, \langle\rangle, s'.\mathtt{refXHRctx}\rangle$
        let $s'.\mathtt{q} := \mathtt{receiveContext}$
        stop $\langle s', cookies', localStorage', sessionStorage, command\rangle$
      end if
    end if
  case `receiveContext`
    let $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
    if $(\pi_1(input) \equiv \mathtt{XMLHTTPREQUEST}) \wedge (\pi_3(input) \equiv s'.\mathtt{refXHRctx})$ **then**
      let $s'.\mathtt{context} := \pi_2(input)$
      if $\pi_1(s'.\mathtt{context}) \equiv \langle\rangle$ **then**
        let $s'.\mathtt{q} := \mathtt{requestAuth}$
        stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
      else
        let $s'.\mathtt{q} := \mathtt{requestUC}$
        stop $\langle s', cookies', localStorage', sessionStorage, \langle\rangle\rangle$
      end if
    end if
  case `requestAuth`
    let $secret \leftarrow secrets$
    let $body := \langle secret, s'.\mathtt{context.xsrfToken}\rangle$
    let $s'.\mathtt{refXHRauth} := \lambda_2$
    let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{auth}}, \mathtt{POST}, body, s'.\mathtt{refXHRauth}\rangle$
    let $s'.\mathtt{q} := \mathtt{receiveContext}$
    stop $\langle s', cookies', localStorage', sessionStorage, command\rangle$

This algorithm is continued on the next page.

```
42:     case receiveAuth
43:         let input, s' := CHOOSEINPUT(s', scriptinputs)
44:         if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRauth) then
45:             if π2(input) ≡ ⊤ then
46:                 let command := ⟨XMLHTTPREQUEST, URL^LPO_/ctx, GET, ⟨⟩⟩
47:                 let s'.q := receiveContext
48:                 stop ⟨s', cookies', localStorage', sessionStorage, command⟩
49:             end if
50:         end if
51:     case requestUC
52:         let id ← ids
53:         let localStorage'[siteInfo][s'.requestOrigin] := id
54:         let s'.key := λ3
55:         let body := ⟨id, pub(s'.key), s'.context.xsrfToken⟩
56:         let s'.refXHRcert := λ4
57:         let command := ⟨XMLHTTPREQUEST, URL^LPO_/certreq, POST, body, s'.refXHRcert⟩
58:         let s'.q := receiveUC
59:         stop ⟨s', cookies', localStorage', sessionStorage, command⟩
60:     case receiveUC
61:         let input, s' := CHOOSEINPUT(s', scriptinputs)
62:         if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRcert) then
63:             let uc := π2(input)
64:             let ia := sig(s'.requestOrigin, s'.key)
65:             let cap := ⟨uc, ia⟩
66:             let command := ⟨POSTMESSAGE, OPENERWINDOW(tree, docnonce),
                  ↪ ⟨response, cap⟩, s'.requestOrigin⟩
67:             let s'.q := null
68:             stop ⟨s', cookies', localStorage', sessionStorage, command⟩
69:         end if
70: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
```

$q \equiv$ `init` is the initial state. It creates the CIF iframe and then transitions to `receiveCIFReady`.

$q \equiv$ `receiveCIFReady` expects a `cifready` postMessage from the CIF iframe with origin of LPO. It chooses some identity, $\langle\rangle$, or $\perp$ and sends this as a `loaded` postMessage to the CIF iframe with receiver's origin set to the origin of LPO.[4] It then transitions to the state `default`.

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

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

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

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

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

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

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

$q \equiv$ `sendCAP` sends the CAP to RP as a `POST` XHR and then takes `receiveServiceToken` as the next state.

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

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

We formally specify the relation *script_RP_index* $\subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}$ of the RP-Doc's scripting process in Algorithm E.5.

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

E.2.9. Important Events

As described in Section 3.2, we define events that (if they occur in a certain processing step) specify important actions of our SSO protocol.

Definition 93 (Refinement of Definition 5 for BrowserID in Secondary Mode: Start of an SSO Flow). Let $\mathcal{BID}^s$ be an BrowserID secondary mode Web system. Let ρ be a run of $\mathcal{BID}^s$. Let $Q \in \rho$ be a processing step, b a browser, and r an RP. We write $\mathsf{started}(Q,b,r)$ iff in Q, the browser b is triggered and selects to run the script of a document and this script is `script_rp_index` and the document is stored in b under a secure origin[5] of r and — when executing the script — in Line 27 of Algorithm E.5 the value `openLD` is (non-deterministically) chosen. ◇

Definition 94 (Refinement of Definition 6 for BrowserID in Secondary Mode: Selection of an IdP). Let $\mathcal{BID}^s$ be an BrowserID secondary mode Web system. Let ρ be a run of $\mathcal{BID}^s$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, and i an IdP. We write $\mathsf{selectedIdP}(Q,b,i)$ iff $\mathsf{started}(Q,b,r)$. Note that i is always LPO. ◇

Definition 95 (Refinement of Definition 7 for BrowserID in Secondary Mode: Selection of an Identity). Let $\mathcal{BID}^s$ be an BrowserID secondary mode Web system. Let ρ be a run of $\mathcal{BID}^s$. Let $Q \in \rho$ be a processing step, b a browser, i an IdP, and id an identity. We write $\mathsf{selectedID}(Q,b,i,id)$ iff in Q, the browser b is triggered and selects to run the script `script_LPO_ld` in some document and — in that script — in Lines 51ff. of Algorithm E.4, id is selected from the browser's identities. ◇

[5] Note that we assume a secure origin of RP here as — similar to reality — we cannot have any integrity of scripts and their states under an insecure origin.

Algorithm E.5: Relation of *script_RP_index*.

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

1: **let** $s' := scriptstate$
2: **let** $cookies' := cookies$
3: **let** $localStorage' := localStorage$
4: **switch** $s'.\mathtt{q}$ **do**
5: **case** `init`
6: **let** $command := \langle \mathtt{IFRAME}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{cif}}, \mathsf{GETWINDOW}(tree, docnonce)\rangle$
7: **let** $s'.\mathtt{q} := \mathtt{receiveCIFReady}$
8: **let** $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
9: **let** $s'.\mathtt{CIFindex} := |subwindows| + 1$ → Index of the next subwindow to be created.
10: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$
11: **case** `receiveCIFReady`
12: **let** $input, s' := \mathsf{CHOOSEINPUT}(s', scriptinputs)$
13: **if** $\pi_1(input) \equiv \mathtt{POSTMESSAGE}$ **then**
14: **let** $senderWindow := \pi_2(input)$
15: **let** $senderOrigin := \pi_3(input)$
16: **let** $m := \pi_4(input)$
17: **let** $subwindows := \mathsf{SUBWINDOWS}(tree, docnonce)$
18: **if** $(m \equiv \langle \mathtt{cifready}, \langle\rangle\rangle)$
↪ $\wedge(senderOrigin \equiv \mathsf{origin}_{\mathsf{LPO}})$
↪ $\wedge(senderWindow \equiv \pi_{s'.\mathtt{CIFindex}}(subwindows).\mathtt{nonce})$ **then**
19: **let** $id \leftarrow \{\bot, \langle\rangle\} \cup \mathsf{ID}$
20: **let** $command := \langle \mathtt{POSTMESSAGE}, \pi_{s'.\mathtt{CIFindex}}(subwindows),$
↪ $\langle \mathtt{loaded}, id\rangle, \mathsf{origin}_{\mathsf{LPO}}\rangle$
21: **let** $s'.\mathtt{q} := \mathtt{default}$
22: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$
23: **end if**
24: **end if**
25: **case** `default`
26: **if** $s'.\mathtt{dialogRunning} \equiv \bot$ **then**
27: **let** $choice \leftarrow \{\mathtt{openLD}, \mathtt{handlePM}\}$
28: **else**
29: **let** $choice := \mathtt{handlePM}$
30: **end if**
31: **if** $choice \equiv \mathtt{openLD}$ **then**
32: **let** $s'.\mathtt{dialogRunning} := \top$
33: **let** $command := \langle \mathtt{HREF}, \mathsf{URL}^{\mathsf{LPO}}_{/\mathtt{ld}}, \mathtt{\_BLANK}\rangle$
34: **let** $s'.\mathtt{q} := \mathtt{default}$
35: **stop** $\langle s', cookies', localStorage', sessionStorage, command\rangle$

This algorithm is continued on the next page.

```
36:         else
37:             let input, s' := CHOOSEINPUT(s', scriptinputs)
38:             if π1(input) ≡ POSTMESSAGE then
39:                 let senderWindow := π2(input)
40:                 let senderOrigin := π3(input)
41:                 let m := π4(input)
42:                 let subwindows := SUBWINDOWS(tree, docnonce)
43:                 if senderOrigin ≡ origin_LPO then
44:                     if senderWindow ≡ π_{s'.CIFindex}(subwindows).nonce then
45:                         if π1(m) ≡ login then
46:                             let s'.cap := π2(m)
47:                             let s'.q := sendCAP
48:                             stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
49:                         else if π1(m) ≡ logout then
50:                             let s'.q := default
51:                             stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
52:                         end if
53:                     else if s'.dialogRunning ≡ ⊤ then
54:                         if π1(m) ≡ ldready then
55:                             let command := ⟨POSTMESSAGE,
                                    ↪ AUXWINDOW(tree, docnonce), ⟨request, ⟨⟩⟩, origin_LPO⟩
56:                             let s'.q := default
57:                             stop ⟨s', cookies', localStorage', sessionStorage, command⟩
58:                         else if π1(m) ≡ response then
59:                             let s'.dialogRunning := ⊥
60:                             let s'.cap := π2(m)
61:                             let command := ⟨CLOSE, AUXWINDOW(tree, docnonce)⟩
62:                             let s'.q := dlgClosed
63:                             stop ⟨s', cookies', localStorage', sessionStorage, command⟩
64:                         end if
65:                     end if
66:                 end if
67:             end if
68:         end if
69:     case dlgClosed
70:         let subwindows := SUBWINDOWS(tree, docnonce)
71:         let id := π1(extractmsg(π1(s'.cap)))    → Extract id from CAP.
72:         let command := ⟨POSTMESSAGE, π_{s'.CIFindex}(subwindows).nonce,
              ↪ ⟨loggedInUser, id⟩, origin_LPO⟩
73:         let s'.q := loggedInUser
74:         stop ⟨s', cookies', localStorage', sessionStorage, command⟩
75:     case loggedInUser
76:         let subwindows := SUBWINDOWS(tree, docnonce)
77:         let command :=
              ↪ ⟨POSTMESSAGE, π_{s'.CIFindex}(subwindows).nonce, ⟨dlgCmplt, ⟨⟩⟩, origin_LPO⟩
78:         let s'.q := sendCAP
79:         stop ⟨s', cookies', localStorage', sessionStorage, command⟩
```

This algorithm is continued on the next page.

```
80:     case sendCAP
81:         let s'.refXHRcap := λ1
82:         let host, protocol such that
              ↪ ⟨host, protocol⟩ ≡ GETORIGIN(tree, docnonce)
              ↪ if possible; otherwise stop
              ↪ ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
83:         let command := ⟨XMLHTTPREQUEST, ⟨URL, protocol, host, /, ⟨⟩⟩, POST, s'.cap,
              ↪ s'.refXHRcap⟩    → Relay received CAP to RP.
84:         let s'.q := receiveServiceToken
85:         stop ⟨s', cookies', localStorage', sessionStorage, command⟩
86:     case receiveServiceToken
87:         let input, s' := CHOOSEINPUT(s', scriptinputs)
88:         if (π1(input) ≡ XMLHTTPREQUEST) ∧ (π3(input) ≡ s'.refXHRcap) then
89:             let s'.q := default
90:             stop ⟨s', cookies', localStorage', sessionStorage, ⟨⟩⟩
91:         end if
92: stop ⟨scriptstate, cookies, localStorage, sessionStorage, ⟨⟩⟩
```

Definition 96 (Refinement of Definition 8 for BrowserID in Secondary Mode: User is Logged in at RP). Let $\mathcal{BID}^s$ be an BrowserID secondary mode Web system. Let ρ be a run of $\mathcal{BID}^s$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, id an identity, and n some term. We write $\mathsf{loggedIn}(Q, b, r, id, n)$ iff in Q, b is triggered, selects a document containing the script `script_rp_index` under a secure origin of r and — in that script — in Lines 88ff. of Algorithm E.5 a response to an XHR is processed that contains n in its body, while in the state of r (before and after Q — note that the state of r cannot be altered during Q), for the subterm *serviceTokens*, we have $serviceTokens[n] \equiv id$. ◇

E.3. Proof of Theorem 3 (Security w.r.t. Authentication)

In order to prove Theorem 3, we have to prove that the BrowserID secondary mode web system satisfies security w.r.t. authentication (Definition 12).

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

Without loss of generality, we may assume that ρ also satisfies the following:

(**) Whenever a browser becomes corrupted (i.e., either `FULLCORRUPT` or `CLOSECORRUPT`) in a processing step leading to some state s_l in ρ, this browser is triggered immediately afterwards again (in the processing step leading to s_{l+1}) and sends the full state of the web browser to the

attacker process attacker, which then receives this knowledge in state s_{l+2}. Afterwards, this browser is not triggered anymore.

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

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

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

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

$$req_{\text{cap}} = \mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, n_{\text{cap}}, \mathtt{POST}, d_r, /, \langle\rangle, [\mathtt{Origin} : \langle d_r, \mathtt{S}\rangle], c\rangle, k\rangle, \mathsf{pub}(\mathsf{tlskey}(d_r))) \ , \tag{E.2}$$

$$resp_{\text{cap}} = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n_{\text{cap}}, 200, \langle sts\rangle, \langle n,i\rangle\rangle, k) \ . \tag{E.3}$$

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

$$s_{m-1} \xrightarrow[r \to \{(x:a_r:resp_{\text{cap}})\}]{(a_r:x:req_{\text{cap}}) \to r} s_m \ .$$

From the assumption and the definition of RPs it follows that c is issued for d_r (otherwise, RP would not accept the CAP, see Line 13 of Algorithm E.2). The nonce n in $\langle n,i\rangle$ is chosen freshly and from RPs nonces N^r. It is not used again by r afterwards.

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

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

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

Lemma 8. In a run ρ of $\mathcal{BID}^s$, if LPO sends a `browserid_state` cookie in a `Set-Cookie` header in an HTTPS response to an HTTPS request emitted by a browser b, there is no state in

the run where the browser is honest and the attacker can derive the cookie value from its own knowledge.

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

The `browserid_state` cookie is sent to b as an httpOnly secure session cookie (compare Line 38 in Algorithm E.1). When the response arrives at b, the cookie is transferred to the cookie store (Line 4 of Algorithm A.8) which is indexed by domains. The cookie information can be accessed by scripts (Line 3 of Algorithm A.7) and can be added to requests (Line 5 of Algorithm A.4). As the `browserid_state` cookie is an httpOnly cookie we can rule out the first case. In the second case, the cookie can only be added to requests to the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$, as the cookie is marked as secure (as defined in Line 5 in Algorithm A.4). These properties hold as long as the browser is not corrupted.

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

First, we can see that the cookie value was either in the browser's knowledge before it received the `browserid_state` header or that it was chosen freshly by LPO. The only line where LPO sets the cookie is in Line 38 of Algorithm E.1. From the lines before, it is easy to see that the session value that finally becomes the cookie value was either provided as a cookie in the request or is chosen from the set of unused nonces. In Lemma 10 we see that any value that is contained in a request sent by an honest browser to LPO was initially chosen by LPO.

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

Lemma 9. In every state $s = (S, E)$ of run ρ of $\mathcal{BID}^s$, for every *xsrfToken* of an LPO session and its session ID *sessionid*, if $\mathit{xsrfToken} \in d_{N^{\mathsf{attacker}}}(S(\mathsf{attacker}))$, then we have that $\mathit{sessionid} \in d_{N^{\mathsf{attacker}}}(S(\mathsf{attacker}))$, i.e., an attacker can only know an *xsrfToken* value for an LPO session if he knows the session ID `browserid_state` of that session.

Proof. The *xsrfToken* is chosen by LPO (Line 34 in Algorithm E.1). If LPO receives a POST request with the path `/ctx` that contains a `browserid_state` cookie containing a *sessionid* that is in its list of valid sessions, it returns *xsrfToken* as part of the response. If LPO receives a request to the same URL without a valid session ID, it creates a new session and returns *sessionid* as well as a freshly chosen *xsrfToken* in the response. For other requests (to other URLs, etc.) *xsrfToken* is not a part of the response at all.

The *xsrfToken* is only transferred over HTTPS: LPO only reacts to HTTPS requests (Line 20 of Algorithm E.1) and the request that is sent from the browser to LPO to retrieve *xsrfToken* is explicitly sent over HTTPS (Line 44 of Algorithm E.3 or Line 18 of Algorithm E.4). Thus, if an honest browser sends a request to LPO, the attacker cannot read the response if the browser stays honest (Lemma 1). If the browser becomes corrupted later, the attacker learns the *sessionid* and the *xsrfToken* at the same time. The LPO script that has access to *xsrfToken* in the browser's state does not sent out this part of the state to origins other than LPO's (see Algorithm E.3 and Algorithm E.4) and the *xsrfToken* is stored only temporarily in the scriptstate (as part of the *context*, see Line 50 in Algorithm E.3 and Line 26 in Algorithm E.4), such that it is never released when the browser is honest or closecorrupted.

We can see that the attacker knows *sessionid* whenever he knows *xsrfToken*, which proves the lemma. □

Lemma 10. In a run ρ of $\mathcal{BID}^s$, for any HTTPS request *req* that is emitted by an honest browser b and that is encrypted with the public key of LPO, if there is a `Cookie` header in *req* containing a cookie with the name `browserid_state`, then there is an HTTPS response that was emitted by LPO previously in the run and that was accepted by b. In this response, a `Set-Cookie` header was sent with the name `browserid_state` and the same value as the `browserid_state` cookie in *req*.

Proof. The cookie that is sent in *req* was taken from the cookie list that is stored in the browser state (see Algorithm A.4). Cookies are stored per-domain, i.e., $\mathsf{dom}(\mathsf{LPO})$ in this case. Adding a cookie to this list can be achieved by adding a `Set-Cookie` to a response on a request to $\mathsf{dom}(\mathsf{LPO})$ or by setting the cookie from a script in a document with the origin $\langle \mathsf{dom}(\mathsf{LPO}), x \rangle$ where $x \in \{\mathtt{P}, \mathtt{S}\}$. The domain $\mathsf{dom}(\mathsf{LPO})$ is part of the *sts* list in honest browsers (see Section 2.5) thus the browser b never contacts the insecure origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{P} \rangle$. Thus, responses and scripts can only be received from the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$ (see Lemma 1 Property (4) and Lemma 2). The LPO scripts *script_LPO_cif* and *script_LPO_ld* do not set cookies, thus the only possible way that a cookie can be stored in the browser's list of cookies is when LPO adds a `Set-Cookie` header to a HTTPS response. Obviously, this header has to have the same value as the cookie that is finally returned to the server. This proves the lemma. □

Lemma 11. In a run ρ of $\mathcal{BID}^s$, if an honest browser b emits a request req_{auth} that is received by LPO and leads to the authentication[6] of an LPO session identified by the sessionid *sessionid*, then the identity i, for which the session was authenticated, is owned by b, i.e., $i \in \mathsf{ownerOfID}^{-1}(b)$.

[6] See Appendix E.2.6 for an explanation on the authentication at LPO.

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

$$\begin{aligned} req_{\text{auth}} = \mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, n_2, \texttt{POST}, \mathsf{dom}(\mathsf{LPO}), /\texttt{auth}, \langle\rangle, \\ \langle[\texttt{Cookie} : [\texttt{browserid\_state} : \mathit{sessionid}], \ldots]\rangle, \\ \langle s, \mathit{xsrfToken}\rangle, \rangle, k''\rangle, \mathsf{pub}(\mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO})))) \end{aligned} \tag{E.4}$$

The request req_{auth} contains the secret s and the *xsrfToken* that, by definition of LPO, is stored at LPO along with *sessionid*. In an honest browser (which b is), this request can only be caused by a script (or through a redirection, which again would require a script to initiate the request in the first place). There are three scripts that can issue such a request: the attacker script and both LPO scripts. In the latter case, the LPO scripts will provide the browser secret as s and hence, authenticate for an identity owned by the browser. In the former case, the attacker script needs to know *xsrfToken*. Hence, by Lemma 9 he needs to know *sessionid*. However, the *sessionid* value does not leak from the honest browser b (Lemma 8) and cannot be set by the attacker (Lemma 10). Hence, the attacker cannot know *sessionid*, and hence, by Lemma 9 he cannot know *xsrfToken*, and hence, req_{auth} cannot have been initiated by the attacker script. □

Lemma 12. In a run ρ of $\mathcal{BID}^s$ if either *script_LPO_cif* or *script_LPO_ld* were loaded into a document with HTTPS origin and are used to create a CAP c, i.e., if c is contained in a postMessage that is sent in Line 82 of Algorithm E.3 or in Line 68 of Algorithm E.4, then the origin for which c is issued is the origin of the script that receives this postMessage.

Proof. Looking at the case when *script_LPO_cif* issues the CAP in Line 80 of Algorithm E.3, the origin for which the IA is issued in this case is determined by the element $s'.\texttt{parentOrigin}$ of the scriptstate. This element is only written to in Line 18 of Algorithm E.3. Its value is the sender origin of the postMessage requesting the CAP. The very same value determines the only allowed receiver origin of the postMessage that returns the CAP (Line 80). With a very similar argument (different line numbers), we can see that the statement for *script_LPO_ld* holds true as well. □

Lemma 13. In a run ρ of $\mathcal{BID}^s$, if a CAP $c = \langle uc, ia\rangle$ is sent by *script_RP_index* (Line 85 of Algorithm E.5) running in an honest browser $b \in \mathsf{B}$ in a document with origin $\langle d_r, \mathsf{S}\rangle$ as an HTTP(S) message to an RP $r \in \mathsf{RP}$, where $d_r = \mathsf{dom}(r)$, $uc = \mathsf{sig}(\langle i, \mathsf{pub}(k_u)\rangle, k^{\mathsf{LPO}})$, and $ia = \mathsf{sig}(o, k'_u)$, then all of the following statements are true:

1. c is a valid CAP. In particular, $k_u = k'_u$.

2. *uc* was created by LPO and transferred to *script_RP_index* in a postMessage by a script of LPO running in b (either *script_LPO_ld*, postMessage sent in Line 82 of Algorithm E.4 or *script_LPO_cif*, postMessage sent in Line 68 of Algorithm E.3) loaded into a document with the origin $\langle \mathsf{dom}(\mathsf{LPO}), \mathtt{S} \rangle$.

3. *ia* contains the origin $o = \langle d_r, \mathtt{S} \rangle$.

4. k_u is not known to any atomic DY process except for b, as long as b is not fullycorrupted.

5. *uc* is issued for an identity $i \in \mathsf{ownerOfID}^{-1}(b)$.

Proof. As we know that the message is sent from the origin $\langle d_r, \mathtt{S} \rangle$, we know that the script *script_RP_index* was loaded over an HTTPS origin (see Lemma 3). Its script state cannot be manipulated by scripts loaded from a different origin (see Algorithms A.2 and A.7).

The only transitions of the script *script_RP_index* which can send out a request to r are the ones starting out from state $scriptstate.\mathtt{q} = \mathtt{sendCAP}$ (Line 80 of Algorithm E.5). These transitions take the CAP c from $scriptstate.\mathtt{cap}$. The only transitions before which could have written something into this place in the scriptstate are the ones where $scriptstate.\mathtt{q} = \mathtt{default}$ (Line 25) when handling a postMessage from origin LPO (we can overapproximate here by ignoring all other side restrictions of this transition, e.g. having $scriptstate.\mathtt{dialogRunning} = \top$). This means that the CAP c was sent by a script with origin LPO. Since origin LPO is also an HTTPS origin, the script must have been sent by LPO (Lemma 2).

The postMessage that was received by *script_RP_index* is checked to be a sequence with the first element being `login` or `response`. Such postMessages are issued by LPO only in Line 80 of Algorithm E.3 (*script_LPO_cif*) and in Line 66 of Algorithm E.4 (*script_LPO_ld*). In both cases, *ia* is signed using the private key k_u that is taken from the respective scriptstate. This element of the scriptstate is only written to once, and with a freshly chosen nonce (Line 68 of Algorithm E.3 and Line 54 of Algorithm E.4, respectively). In both cases, starting in the state `requestUC`, an XHR to LPO is sent to have $\mathsf{pub}(k_u)$ signed by LPO. From the response to this request, *uc* is extracted. The request is always sent over HTTPS to LPO. Lemma 1, in particular Property (4), applies. Therefore, we see that *uc* was actually sent by LPO.

In the definition of LPO (Line 60 of Algorithm E.1) we see that LPO only sends out freshly created *uc*'s. LPO only issues valid UCs (if any). Once returned to the script *script_LPO_cif* or *script_LPO_ld*, the UC is combined with an IA and sent to *script_RP_index* (which is determined by the sender of the initial CAP request and its origin). This script sends the CAP to r. Thus, the CAP that is sent is always valid, which proves (1). Further, the UC was always created by LPO, proving (2).

Property (3) follows immediately from the above observations (i.e., the script *script_LPO_cif* or *script_LPO_ld* was loaded over HTTPS and is used to create the CAP) and Lemma 12.

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

Property (5) follows immediately with the observations in the proof of Property (2) and Lemma 11. □

Lemma 14. If in a run ρ of $\mathcal{BID}^s$ an IA *ia* for an origin $\langle d_r, \mathtt{S}\rangle$ where $d_r \in \mathsf{dom}(\mathsf{RP})$ is signed in the scripts *script_LPO_ld* or *script_LPO_cif* in an honest browser b, and these scripts were loaded over HTTPS from LPO, then at most b and the RP $r = \mathsf{dom}^{-1}(d_r)$ know *ia*.

Proof. The scripts *script_LPO_ld* and *script_LPO_cif* send the *ia* to the parent or opener window (respectively) using a postMessage. For this postMessage, the only allowed receiver origin is the same as the origin for which *ia* was issued, so in our case $\langle d_r, \mathtt{S}\rangle$ (see proof for Lemma 13 Property (3)). The script *script_RP_index*, which thus must be the receiver, sends the complete CAP (containing *ia*) to RP using HTTPS. The RP discards the CAP after checking it. The CAP and especially *ia* therefore cannot leak. □

Lemma 15. In a run ρ of $\mathcal{BID}^s$, if LPO creates a message containing a UC *uc* for an identity i of a browser b, then there is no state in the run ρ where b is honest and attacker knows the private key k_u corresponding to the public key $\mathsf{pub}(k_u)$ that was signed in *uc*.

Proof. First, it is easy to see that the Lines 60–63 in Algorithm E.1 have to be used in the transition to create *uc*: At no other point in the definition of LPO *uc* is created or emitted. From Line 54 and following it is easy to see that a request of the following form has to be sent to LPO in order to create *uc*:

$$\begin{aligned} req_{\mathrm{uc}} = \mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, n_1, \mathtt{POST}, \mathsf{dom}(\mathsf{LPO}), /\mathtt{certreq}, \langle\rangle, \\ \langle[\mathtt{Cookie} : [\mathtt{browserid\_state} : \mathit{sessionid}], \ldots]\rangle, \\ \langle i, \mathsf{pub}(k_u), \mathit{xsrfToken}\rangle\rangle, k''\rangle, \mathit{pub}(\mathsf{tlskey}(\mathsf{dom}(\mathsf{LPO})))) \end{aligned} \tag{E.5}$$

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

- **The attacker knows k'':** In this case, we can see with Lemma 1 that no honest browser has created req_{uc}. As RP, LPO, and dishonest browsers do not emit requests in general,

only the attacker can have created this request. For this, he needs to know *xsrfToken* and *sessionid*.

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

The first case can be ruled out, as the attacker cannot know the *sessionid* value (Lemma 8).

In the second case, he cannot create a message that leads to the authentication of the session himself (which would require knowledge of the secret for identity i) and he cannot force the owner browser of i to authenticate the session (Lemma 11).

- **The attacker does not know k'':** In this case, the request was not created by the attacker. As above, RP, LPO and dishonest browsers do not create requests. Thus, this request was created by an honest browser ("honest" in the state when req_{uc} was created) and an honest script in that browser (with a dishonest script, the attacker would need to know *xsrfToken*, which he does not according to Lemma 9 and Lemma 8). If an honest script, i.e., *script_LPO_cif* or *script_LPO_ld*, is used, the attacker does not learn k_u (Lemma 13 Property (4)).

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

□

Lemma 16. If in a run ρ of $\mathcal{BID}^s$ a browser b created the request req_{cap} defined in (E.2), then (i) req_{cap} was sent from the script *script_RP_index* that was loaded over an HTTPS origin from r while the browser was honest or (ii) req_{cap} was encrypted by the attacker script while the browser was honest and the attacker knows the CAP c and the symmetric key k.

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

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

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

In the latter case (script encrypted request), which corresponds to statement (ii) in the lemma, we see that the honest scripts do not encrypt messages and thus, the attacker script is the only

script that can do so. To do so, the script needs to know every component of req_{cap} before the encryption, in particular k and c. These have been sent to the attacker before the encryption according to (***). Thus, the attacker must know k and c before. □

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

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

By our assumption, we know that

$$\langle n,i\rangle \notin d_{N^{\mathsf{attacker}}}(S_{j-1}(\mathsf{attacker}))$$

and that

$$\langle n,i\rangle \in d_{N^{\mathsf{attacker}}}(S_{j-1}(\mathsf{attacker}),m) \ .$$

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

The attacker does not know k in s_j.

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

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

- **The browser that owns i** created req_{cap}: By Lemma 16 it follows that the browser was honest when encrypting req_{cap}, and req_{cap} was initiated by *script_RP_index*, which was delivered over HTTPS from r. Note that we can rule out case (ii) in the lemma, as the attacker does not know k.

  This script initiated req_{cap} and it is easy to see that this script (or no script at all) receives the corresponding response: From the browser definition, we see that XHR responses are delivered to the document with the same nonce as the document that initiated the request (Line 56 in Algorithm A.8). Other documents have no access to the data from this document, except for same-origin documents (this is ensured by the Clean function that is used in Line 2 of Algorithm A.7 and by the GETWINDOW function (Algorithm A.2) that determines the windows which can be manipulated by other scripts). However, other same-origin documents can only contain the script *script_RP_index* (this is the only script that RP sends, and with Lemma 2 we see that other same-origin documents cannot have

been sent by the attacker). Other manipulations to the window of the document (e.g., navigating the window away) change the active document in the window (Algorithm A.8) and could only prevent the script from receiving the response.

From Algorithm E.5 it is easy to see that after $\langle n, i \rangle$ is delivered back to *script_RP_index* after $resp_{\text{cap}}$ was received, nothing happens with $\langle n, i \rangle$: If the browser is uncorrupted, only same-origin scripts have access to it (as shown above), but there are no scripts which use the information. The information can therefore not leak to the attacker. If the browser is closecorrupted before receiving $resp_{\text{cap}}$, the attacker cannot derive $\langle n, i \rangle$ from its information, as the decryption key is lost. If the browser is closecorrupted after receiving $resp_{\text{cap}}$, by definition of close-corruption, $\langle n, i \rangle$ is removed from the browser's state before the browser can be controlled by the adversary. By the assumption in (*), the browser cannot be fullycorrupted at any point in the run. Hence, in contradiction to (*), the attacker cannot obtain $\langle n, i \rangle$.

- A **browser that does not own** i created req_{cap}: In this case, it still holds that the browser was honest when encrypting req_{cap} and the script *script_RP_index* created the request and was loaded over HTTPS (Lemma 16). With Lemma 3 and Lemma 13, in particular Properties (2) and (5), we see that the RP script only initiates HTTPS requests containing CAPs that have been created by LPO and for an identity of the browser. This is in contradiction to the fact that i is not owned by the browser but req_{cap} contains a CAP for i. Hence, req_{cap} cannot have been created by this browser.

- **RPs or LPO** created req_{cap}: As per their definitions (Algorithms E.1 and E.2), they do not initiate or create HTTP(S) requests.

- **The attacker process** created req_{cap}: It is clear that any atomic process that created req_{cap} needs to know k. It follows, by our assumption that the attacker cannot derive k, that the attacker has not created req_{cap}.

The attacker knows k in s_j.

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

- **The browser that owns** i created req_{cap}: By our assumption (*), this browser cannot be fully corrupted in the run. By Lemma 16, it follows in the case (i) that *script_RP_index* sent the request and that k cannot be known by the attacker (with Lemma 1, Property (2)) and hence, the browser cannot have created req_{cap}. In the case (ii) it follows that the attacker needs to know the CAP c in order to create the request.

- **A browser not owning** i created req_{cap}: By Lemma 16, we see that the browser was honest while encrypting req_{cap} and (i) that *script_RP_index* sent the request. With Lemma 13 Property (5) we see that the browser cannot have created req_{cap} because it only creates requests for its own identities. In the case of (ii) we see that, again, the attacker has to know the cap c in order to create the request.

- **RPs or LPO** created req_{cap}: As per their definitions (Algorithms E.1 and E.2), they do not emit HTTP requests.

- **The attacker process** created req_{cap}: It is clear that any atomic process that created req_{cap} needs to know c.

As we can see, the attacker needs to know $c = \langle uc, ia \rangle$ before he is able to create req_{cap}. With Lemma 15 we know that the attacker cannot request uc itself with the identity i, and thus, he cannot know the key k_u that was signed in uc. Neither can any browser other than b know k_u, otherwise the attacker could corrupt this browser and learn k_u. The key k_u is needed to create ia, therefore only in b the identity assertion ia can be created, and it can only be created by *script_LPO_cif* or *script_LPO_ld* (LPO checks the origin of the request for uc, and only the script that sends uc knows k_u). With Lemma 14 we see that the attacker cannot learn ia.

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

E.4. Proof of Theorem 4 (Security w.r.t. Session Integrity)

Similar to the primary mode, in order to prove Theorem 4, we have to proof that the BrowserID secondary mode web system satisfies security w.r.t. indirect session integrity (Definition 13).

We start from a processing step Q_{login} in which the browser is actually logged in and trace back on the causality chain (the session), i.e., for every processing step in the session, we analyze all other possible processing steps that can possibly lead to the later processing steps.

To start, we first need to assume that we have an arbitrary processing step in which the browser gets logged in: Let $\mathcal{BID}^s$ a BrowserID secondary mode web system, ρ an arbitrary run of $\mathcal{BID}^s$, Q_{login} an arbitrary processing step in ρ, b an arbitrary browser of $\mathcal{BID}^s$ that is honest in Q_{login}, r an arbitrary RP of $\mathcal{BID}^s$ that is honest in Q_{login}, and id an arbitrary identity.

To show the main implication of the session integrity property, we now assume that there exists some n such that $\mathsf{loggedIn}(Q_{\text{login}}, b, r, id, n)$. This means that b is currently processing a response (say, $resp_{\text{CAP}}$) to an XHR in a document under a secure origin of r that contains the

script `script_rp_index`. We call the document RP-Doc. Further, b has received an HTTP(S) response before that contains n in its body. This response must have a corresponding HTTP(S) request (say, req_{CAP}) that bears the XHR reference that is stored in the subterm *refXHRcap* in the scriptstate. This reference is only ever set in Lines 80ff. of Algorithm E.5. Hence, this request/response pair must use HTTPS.

We know that req_{CAP} is a POST request that carries some value *cap* in its body. The RP r only processes such a request in Lines 8ff. of Algorithm E.2. There, r checks whether *cap* is a valid CAP (the UC must be signed by LPO and the IA must be signed by the key certified in the UC) and extracts the identity id' for which the CAP was created. If the CAP is valid, an entry in the subterm *serviceTokens* is created such that $serviceTokens[n'] \equiv id'$. The value of n' is output inside an HTTPS response, which is $resp_{\text{CAP}}$. Hence, we have that $n \equiv n'$ and $id \equiv id'$ (as the subterm *serviceTokens* in the state of r is never changed, except by adding entries indexed by a fresh nonce).

Recall that the script `script_rp_index` is implemented as a state machine with the state stored in the subterm q of its state. The request req_{CAP} above is only ever created by this script in Lines 80ff. (of Algorithm E.5), which are only executed if the state machine is in state `sendCAP`. In these lines, the CAP is retrieved from the subterm *cap* in the scriptstate.

We now continue by analysing in which cases RP-Doc gets into the state `sendCAP` and how *cap* in the scriptstate is set. To reach the state `sendCAP`, there are only two possibilities:

Lines 75ff. In this case, RP-Doc must have been in state `loggedInUser`, which is only ever reached if Lines 69ff. have been processed in an earlier processing step. This state, in turn, can only be reached if the Lines 58ff. have been processed in an earlier processing step (say Q_{rpcap}). In these lines, also *cap* is stored in the state and there cannot be a different processing step after this processing step (up to Q_{login} in which this subterm is changed. In Q_{rpcap}, the value for *cap* is taken from a postMessage of the form $\langle \texttt{response}, cap \rangle$. This postMessage must have been sent by a document of LPO's origin.

LPO only ever delivers the scripts `script_LPO_cif` and `script_LPO_ld` out of which only `script_LPO_ld` ever creates such a postMessage. This postMessage is created in Line 66 of Algorithm E.4. We call the document in which this step is performed LD-Doc and the processing step in which this postMessage is created $Q_{\text{sendresponse}}$. Recall that the script `script_LPO_ld` is also implemented as a state machine with the state stored in the subterm q of the scriptstate. In LD-Doc, the postMessage above is only created if LD-Doc is in the state `receiveUC` and a response to an XHR is processed that is referenced by the value contained in the subterm *refXHRcert* of the LD-Doc's state. The value of *refXHRcert* is only set in Lines 51ff., where an XHR is is created. This part of the script is also the only

place in which LD-Doc is set to the state `receiveUC`. We call the processing step in which this part of the script is processed $Q_{\text{selectedID}}$. Also note that *refXHRcert* is set to a fresh nonce. Hence, there can only be one XHR of which the response is processed in the state `receiveUC`. We call the request and response of this XHR (from b's perspective) req_{UC} and $resp_{\text{UC}}$, respectively. This XHR is sent as a POST request to the (secure) origin of LPO and the path `certreq`. The body is of the form $\langle id'', \texttt{pub}(s'.\texttt{key}), s'.\texttt{context.xsrfToken}\rangle$. The value for id'' is chosen non-deterministically from *ids*. Hence, we have for the processing step $Q_{\text{selectedID}}$ that $\mathsf{selectedID}(Q_{\text{selectedID}}, b, \mathsf{LPO}, id'')$ and that id'' is in fact an identity that is owned by b.

We know that req_{UC} is processed at LPO in Lines 54ff. of Algorithm E.1 and that LPO creates the corresponding response $resp_{\text{UC}}$. At this place at LPO, the value for the UC *uc* is created for the identity id'' and the body of the response is set to *uc*. Hence, at LD-Doc, in processing step $Q_{\text{sendresponse}}$, in Line 79 of Algorithm E.3, the *cap* is created using this exact value of *uc*. As *cap* is passed on to r (see tracing above), and r uses id'' to determine the identity for the service session, we have that $id \equiv id''$.

We now further trace back our run ρ to the user's selection to actually start the login. We have that the subterm *dialogRunning* of RP-Doc's state must be $\top$ in Q_{rpcap}. There is only one place in the script `script_rp_index` at which this subterm is set to $\top$. (In its initial state, this subterm is always $\bot$.) This change is performed in Line 32 of Algorithm E.5. We call the processing step in which this selection is done Q_{start}. We have that in Q_{start}, the script of RP-Doc must have chosen the value `openLD` in Line 27. Hence, we have that $\mathsf{started}(Q_{\text{start}}, b, r)$ and as LPO is the only IdP in the secondary mode, we have that $\mathsf{selectedIdP}(Q_{\text{start}}, b, \mathsf{LPO})$ and there cannot be any other $Q', i' \neq \mathsf{LPO}$ with $\mathsf{selectedIdP}(Q', b, i')$.

Therefore, we have that $\exists o \in \mathsf{SSOSessions}(\rho, b, r)$ with $Q_{\text{start}} \in o$ and $Q_{\text{selectedID}} \in o$.

Lines 46ff. In this case, RP-Doc must have been in the state `default` and must have chosen to process a postMessage that was sent from a (secure) origin of LPO and from a window identified by the subterm *CIFindex* of the scriptstate. This subterm is initially $\bot$ and only ever set in Line 6ff., where it is set to the identifier of a freshly created iframe. We know that the postMessage is of the form $\langle \texttt{login}, cap' \rangle$. As above, there is only one script delivered by LPO that creates such a postMessage: `script_LPO_cif`. We call this document CIF-Doc here. In this script, such a postMessage is only created in Lines 80ff. of Algorithm E.3. Analogously to above, the script (again modelled as a state machine, similar to `script_LPO_ld`) must be in the state `receiveUC` and is processing a response to an XHR that is identified by the reference stored in the subterm *refXHRcert* of the

scriptstate. This reference is only set in Lines 66ff., which is also the only place where the state can transition to `receiveUC`. Analogously to above, a UC is requested at LPO for some identity id'''. The value for id''' is taken from the localStorage under the index of this document's parent origin. The parent of CIF-Doc, however, is RP-Doc in this case. Therefore, the value of id''' is stored under some secure origin of r. This part of the localStorage under the (secure) origin of LPO is only ever set in Line 53 of Algorithm E.4.

We now know, that `script_LPO_ld` must have run with the same secure origin of r (stored in the subterm *requestOrigin* in its state). This subterm is only ever set in Line 16 of Algorithm E.4 (note that the scriptstate does not return to the state `start`, hence, this line is only executed at most once for one instance of the document). There, it is set to the origin of the sender of a postMessage of the form $\langle \texttt{request}, \langle\rangle \rangle$. Hence, a document containing `script_rp_index` under the secure origin of r as above must have sent this postMessage. This is only done in Lines 16ff. of Algorithm E.5. This line can only be reached if the subterm *dialogRunning* of this scriptstate is $\top$. With the same reasoning as above (i.e., the user must have completed the LD), we have that there exists a processing step Q'_{login} with $\mathsf{started}(Q_{\text{login}}, b, r)$ forming an SSO session o', there exists a $Q'_{\text{selectedID}} \in o'$ with $\mathsf{selectedID}(Q'_{\text{selectedID}}, b, LPO, id''')$, fulfilling our property for indirect session integrity.

■

F. Model and Analysis of SPRESSO

F.1. Formal Model of SPRESSO

We here complete our formal model of SPRESSO outlined in Section 5.3. 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 an SSO Web system (in the sense of Section 3.1). We call an SSO Web system $\mathcal{SWS} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ an *SPRESSO Web system* if it is of the form described in what follows.

Recall that the system $\mathcal{W}^{sp} = \mathsf{Hon} \dot\cup \mathsf{Web} \dot\cup \mathsf{Net}$ consists of Web attacker processes (in Web), network attacker processes (in Net), a finite set FWD (= Other) 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 $\mathsf{Hon} := \mathsf{B} \dot\cup \mathsf{RP} \dot\cup \mathsf{IDP} \dot\cup \mathsf{DNS} \dot\cup \mathsf{FWD}$. More details on the processes in $\mathcal{W}^{sp}$ are provided below. Figure F.1 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.8.

We will now define the DY processes in $\mathcal{SWS}$ and their addresses, domain names, and secrets in more detail. The scripts are defined in detail in Appendix F.1.10.

F.1.1. 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.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_rp	`script_rp`
script_idp	`script_idp`
script_fwd	`script_fwd`

Table F.1.: List of scripts in $\mathcal{S}$ and their respective string representations for SPRESSO.

F.1.2. Keys and Secrets

The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. Atomic processes are given the private keys for their domain in the following form: For an atomic DY process p we define

$$tlskeys^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle \ .$$

The set K_{sign} contains the keys that will be used by IdPs for signing IAs. Let $\mathsf{signkey}\colon \mathsf{IdPs} \to K_{\text{sign}}$ 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.

F.1.3. Corruption

RPs and IdPs 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 or an IdP is *honest* if the according part of their state ($s.\texttt{corrupt}$) is $\bot$, 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.

F.1.4. Attackers

Each $wa \in \mathsf{Web}$ is a Web attacker and each $na \in \mathsf{Net}$ is modeled to be a network attacker as specified in Appendix A.8. The initial state of each (Web or network) attacker a is $s_0^a = \langle attdoms, tlskeys, signkeys \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by *wa*, *tlskeys* is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all verification keys for all IdPs. Web attackers are given an IP address (disjoint from IP address of other processes) and network attackers are allowed to listen to/spoof all available IP addresses, and hence, we define $I^{na} = \mathsf{IPs}$ as usual.

Note that in a setting with a network attacker, all other parties use the attacker as a DNS server. In a system for privacy analysis, however, certain processes use an honest DNS server (see Section F.1.12 below).

F.1.5. Browsers

Each $b \in \mathsf{B}$ is a Web browser as defined in Appendix A.6, with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $ID^b := \mathsf{ownerOfID}^{-1}(b)$ 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 $SecretDomains^b := \{d \mid ID^{b,d} \neq \emptyset\}$ 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 (tls) key, according to the mapping tlskey; the DNS address is $\mathsf{addr}(p)$ with $p \in \mathcal{W}^{sp}$; the list of secrets contains an entry $\langle\langle d, \mathsf{S}\rangle, s\rangle$ for each $d \in SecretDomains^b$ and $s = \mathsf{secretOfID}(i)$ for some $i \in ID^{b,d}$ (s is the same for all i); *ids* is $\langle ID^b\rangle$; *sts* is empty.

F.1.6. Identity Providers

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

Similar to IdPs in BrowserID's primary mode, we will 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 identites $id_1$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 $userset^i = \langle\{\langle u, \mathsf{secretOfID}(u)\rangle \mid u \in \mathsf{ID}^i\}\rangle$.

Definition 97. A *state $s \in Z^i$ of an IdP i* is a term of the form $\langle$*tlskeys*, *users*, *signkey*, *sessions*, *corrupt*$\rangle$ where $tlskeys = tlskeys^i$, $users = userset^i$, $signkey \in \mathcal{N}$ (the key used by the IdP i to sign UCs), $sessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$.

An *initial state s_0^i of i* is a state of the form $\langle tlskeys^i, userset^i, \mathsf{signkey}(i), \langle\rangle, \bot\rangle$. ◇

The relation R^i that defines the behavior of the IdP i is defined in Algorithm F.1.

Algorithm F.1: Relation of an IdP R^i.

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

let $s' := s$
if $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
  let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
end if
let m_{dec}, k, k', *inDomain* **such that**
  $\hookrightarrow$ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{tlskeys}$
  $\hookrightarrow$ **if possible; otherwise stop**
let n, *method*, *path*, *parameters*, *headers*, *body* **such that**
  $\hookrightarrow$ $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
  $\hookrightarrow$ **if possible; otherwise stop**
if *path* $\equiv$ /.well-known/spresso-info **then** $\rightarrow$ Serve support document.
  let *wkDoc* $:= \langle\langle \texttt{signkey}, \mathsf{pub}(s'.\texttt{signkey})\rangle\rangle$
  let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \mathit{wkDoc}\rangle, k)$
  stop $\langle\langle f, a, m'\rangle\rangle$, s'
else if *path* $\equiv$ /.well-known/spresso-login **then** $\rightarrow$ Serve login dialog.
  let *sessionid* $:=$ *headers*[Cookie][sessionid]
  let *email* $:= s'.\texttt{sessions}[\mathit{sessionid}]$
  let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{script\_idp}, \langle \texttt{start}, \mathit{email}, \langle\rangle\rangle\rangle\rangle, k)$
    $\rightarrow$ Initial scriptstate of *script_idp* (defined below).
  stop $\langle\langle f, a, m'\rangle\rangle$, s'
else if *path* $\equiv$ /sign $\wedge$ *method* $\equiv$ POST **then** $\rightarrow$ Serve signing request.
  let *sessionid* $:=$ *headers*[Cookie][sessionid]
  let *loggedInAs* $:= s'.\texttt{sessions}[\mathit{sessionid}]$
  if *body*[email] $\not\equiv$ *loggedInAs* $\wedge$ *body*[password] $\not\equiv s'.\texttt{userset}[\mathit{body}[\texttt{email}]]$ **then**
    stop
  end if
  let *ia* $:= \mathsf{sig}(\langle \mathit{body}[\texttt{tag}], \mathit{body}[\texttt{email}], \mathit{body}[\texttt{FWDDomain}]\rangle, s'.\texttt{signkey})$
  let *sessionid* $:= v_8$
  let $s'.\texttt{sessions}[\mathit{sessionid}] := \mathit{body}[\texttt{email}]$
  let *setCookie* $:= \langle \texttt{Set-Cookie}, \langle\langle \texttt{sessionid}, \mathit{sessionid}, \top, \top, \top\rangle\rangle\rangle$
  let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle \mathit{setCookie}\rangle, \mathit{ia}\rangle, k)$
  stop $\langle\langle f, a, m'\rangle\rangle$, s'
end if
stop

F.1.7. Relying Parties

A relying party $r \in \mathsf{RP}$ is a Web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$. 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*. Recall that service sessions allow a user to use RP's services. The ultimate goal of a login flow is to establish such a service session.

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 $I^r = \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 98. A *login session record* is a term of the form $\langle email, rpNonce, iaKey, tag \rangle$ with $email \in \mathsf{ID}$ and $rpNonce, iaKey, tag \in \mathcal{N}$. ◇

Definition 99. A *state $s \in Z^r$ of an RP r* is a term of the form $\langle DNSAddress, FWDDomain, keyMapping, tlskeys, pendingDNS, pendingRequests, loginSessions, serviceTokens, wkCache, corrupt \rangle$ where $DNSAddress \in \mathsf{IPs}$, $FWDDomain \in \mathsf{Doms}$, $keyMapping \in [\mathbb{S} \times \mathcal{N}]$, $tlskeys = tlskeys^r$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $serviceTokens \in [\mathcal{N} \times \mathsf{ID}]$, $loginSessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary of login session records, $wkCache \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$.

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

As for BrowserID, we define the function serviceSessions that extracts service sessions from an RP's state as the projection that extracts the subterm *serviceTokens*.

We now specify the relation R^r. For readability, we define the relation R^r of RPs in Algorithm F.3 and the function SENDSTARTLOGINRESPONSE in Algorithm F.2 that can be called in R^r at two different points depending on the content of *wkCache*.

F.1.8. Forwarders

We define forwarders formally as atomic DY processes $fwd = (I^{fwd}, Z^{fwd}, R^{fwd}, s_0^{fwd})$. As already mentioned, we define $I^{fwd} = \mathsf{addr}(fwd)$ with the set of states Z^{fwd} consisting of only the initial

Algorithm F.2: Function for RPs to send a response to a startLogin XHR.

function SENDSTARTLOGINRESPONSE(a, f, k, n, *email*, *inDomain*, s')
  let *rpNonce* := v_1
  let *tagKey* := v_2
  let *iaKey* := v_3
  let *loginSessionToken* := v_4
  let *tag* := $\mathsf{enc_s}(\langle \mathit{inDomain}, \mathit{rpNonce}\rangle, \mathit{tagKey})$
  let s'.`loginSessions`[*loginSessionToken*] := $\langle \mathit{email}, \mathit{rpNonce}, \mathit{iaKey}, \mathit{tag}\rangle$
  let *domain* := *email*.`domain`
  let *params* := $\langle\langle$`email`, *email*$\rangle, \langle$`tag`, *tag*$\rangle, \langle$`iaKey`, *iaKey*$\rangle, \langle$`FWDDomain`, s'.`FWDDomain`$\rangle\rangle$
  let *loginURL* := $\langle$`URL`, `S`, *domain*, `/.well-known/spresso-login`, *params*$\rangle$
  let *body* := $\langle\langle$`tagKey`, *tagKey*$\rangle, \langle$`loginSessionToken`, *loginSessionToken*$\rangle$,
    ↪ $\langle$`FWDDomain`, s'.`FWDDomain`$\rangle, \langle$`loginURL`, *loginURL*$\rangle\rangle$
  let m' := $\mathsf{enc_s}(\langle$`HTTPResp`, n, 200, $\langle\langle$`ReferrerPolicy`, `no-referrer`$\rangle\rangle$, *body*$\rangle, k)$
  stop $\langle\langle f, a, m'\rangle\rangle$, s'
end function

Algorithm F.3: Relation of an RP R^r.

Input: $\langle a, f, m\rangle, s$
if s'.`corrupt` $\not\equiv \bot \vee m \equiv$ `CORRUPT` **then**
  let s'.`corrupt` := $\langle\langle a, f, m\rangle, s'$.`corrupt`$\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow$ IPs
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
end if
if $\exists \langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'$.`pendingRequests`
  ↪ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key}))$ ≡ `HTTPResp` **then** → Encrypted HTTP response
  let m' := $\mathsf{dec_s}(m, \mathit{key})$
  if m'.`nonce` $\not\equiv$ *request*.`nonce` **then**
    stop
  end if
  remove $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** s'.`pendingRequests`
  let a', f', k, n, *email*, *inDomain* **such that** $\langle a', f', k, n, \mathit{email}, \mathit{inDomain}\rangle \equiv \mathit{reference}$ **if possible;**
    ↪ **otherwise stop**
  let s'.`wkCache`[*request*.`host`] := m'.`body`
  SENDSTARTLOGINRESPONSE(a', f', k, n, *email*, *inDomain*, s')
else if $m \in$ DNSResponses **then** → Successful DNS response
  if m.`nonce` $\notin s$.`pendingDNS` $\vee$ m.`result` $\notin$ IPs $\vee$ m.`domain` $\not\equiv \pi_2(s$.`pendingDNS`$)$.`host` **then**
    stop
  end if
  let $\langle \mathit{reference}, \mathit{message}\rangle$:= s.`pendingDNS`[m.`nonce`]
  let s'.`pendingRequests` := s'.`pendingRequests`
    ↪ $+^{\langle\rangle}$ $\langle$*reference*, *message*, v_5, m.`result`$\rangle$
  let *message* := $\mathsf{enc_a}(\langle \mathit{message}, v_5\rangle, s'$.`keyMapping`[*message*.`host`]$)$
  let s'.`pendingDNS` := s'.`pendingDNS` − m.`nonce`
  stop $\langle\langle m$.`result`, a, *message*$\rangle\rangle$, s'
This algorithm is continued on the next page.

25: **else** → Handle HTTP requests
26: **let** m_{dec}, k, k', *inDomain* **such that**
↪ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{tlskeys}$
↪ **if possible; otherwise stop**
27: **let** n, *method*, *path*, *parameters*, *headers*, *body* **such that**
↪ $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
28: **if** $\mathit{path} \equiv /$ **then** → Serve index page.
29: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{script\_rp}, \mathit{initState}_{rp}\rangle\rangle, k)$
→ Initial state defined for *script_rp* (below).
30: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
31: **else if** $\mathit{path} \equiv /\texttt{startLogin} \wedge \mathit{method} \equiv \texttt{POST}$ **then** → Serve start login request.
32: **if** $\mathit{body} \notin \mathit{ids}$ **then**
33: **stop**
34: **end if**
35: **let** *domain* := *body*.domain
36: **if** $\mathit{domain} \in s.\texttt{wkCache}$ **then**
37: SENDSTARTLOGINRESPONSE(a, f, k, n, *body*, *inDomain*, s')
38: **else**
39: **let** $\mathit{message} := \langle \texttt{HTTPReq}, v_6, \texttt{GET}, \mathit{domain}, /\texttt{.well-known/spresso-info}, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
40: **let** $s'.\texttt{pendingDNS}[v_7] := \langle\langle a, f, k, n, \mathit{body}, \mathit{inDomain}\rangle, \mathit{message}\rangle$
41: **stop** $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, \mathit{domain}, v_7\rangle\rangle\rangle$, s'
42: **end if**
43: **else if** $\mathit{path} \equiv /\texttt{login} \wedge \mathit{method} \equiv \texttt{POST}$ **then** → Serve login request.
44: **if** $\mathit{headers}[\texttt{Origin}] \not\equiv \langle \mathit{inDomain}, \texttt{S}\rangle \vee \mathit{body}[\texttt{loginSessionToken}] \equiv \langle\rangle$ **then**
45: **stop**
46: **end if**
47: **let** *loginSession* := $s'.\texttt{loginSessions}[\mathit{body}[\texttt{loginSessionToken}]]$
48: **if** $\mathit{loginSession} \equiv \langle\rangle$ **then**
49: **stop**
50: **end if**
51: **let** $s'.\texttt{loginSessions} := s'.\texttt{loginSessions} - \mathit{body}[\texttt{loginSessionToken}]$
52: **let** $ia := \mathsf{dec_s}(\mathit{body}[\texttt{eia}], \mathit{loginSession}.\texttt{iaKey})$
53: **let** $e := \langle \mathit{loginSession}.\texttt{tag}, \mathit{loginSession}.\texttt{email}, s'.\texttt{FWDDomain}\rangle$
54: **let** $\mathit{signkey} := s'.\texttt{wkCache}[\mathit{loginSession}.\texttt{email.domain}][\texttt{signkey}]$
55: **if** $\mathsf{checksig}(ia, \mathit{signkey}) \equiv \bot \vee e \not\equiv \mathsf{extractmsg}(ia)$ **then**
56: **stop**
57: **end if**
58: **let** *serviceTokenNonce* := v_8
59: **let** *serviceToken* := $\langle \mathit{serviceTokenNonce}, \mathit{loginSession}.\texttt{email}\rangle$
60: **let** $s'.\texttt{serviceTokens} := s'.\texttt{serviceTokens} +^{\langle\rangle} \mathit{serviceToken}$
61: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\langle \texttt{ReferrerPolicy}, \texttt{no-referrer}\rangle\rangle, \mathit{serviceToken}\rangle, k)$
62: **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
63: **end if**
64: **end if**
65: **stop**

Algorithm F.4: Relation of an FWD R^{fwd}.

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

1: **let** m_{dec}, k, k', *inDomain* **such that**
↪ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s$
↪ **if possible; otherwise stop**
2: **let** n, *method*, *path*, *parameters*, *headers*, *body* **such that**
↪ $\langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop**
3: **let** $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{script\_fwd}, \langle\rangle\rangle\rangle, k)$
4: **stop** $\langle\langle f, a, m'\rangle\rangle, s$

state $s_0^{fwd} = tlskeys^{fwd}$, i.e., the initial state contains only the private keys of the forwarder's domains. A forwarder responds to any HTTPS request with `script_fwd` and its initial state, which is empty.

We specify the relation R^{fwd} of FWDs in Algorithm F.4.

F.1.9. DNS Servers

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

F.1.10. SPRESSO Scripts

As already mentioned above, the set $\mathcal{S}$ of the Web system $\mathcal{SWS} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ consists of the scripts R^{att}, *script_rp*, *script_idp*, and *script_fwd*, with their string representations being defined by script (see Table F.1).

In what follows, the scripts *script_rp*, *script_idp*, and *script_fwd* are defined formally.

Relying Party Index Page (script_rp). We first describe the structure of the internal scriptstate of the script *script_rp*.

Definition 100. 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*, $tagKey \in \mathcal{N} \cup \{\bot\}$, $FWDDomain \in \mathcal{T}_{\mathcal{N}}$.

The *initial scriptstate* $initState_{rp}$ of *script_rp* is $\langle \texttt{start}, \bot, \bot, \bot, \bot\rangle$. ◇

We specify the relation *script_rp* formally in Algorithm F.5.

Algorithm F.5: Relation of *script_rp*.

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

let $s' := scriptstate$
let $command := \langle\rangle$
let $origin :=$ GETORIGIN$(tree, docnonce)$
switch $s'.\texttt{q}$ **do**
  case `start`
    let $s'.\texttt{email} \leftarrow ids$
    let $s'.\texttt{refXHR} := \lambda_1$
    let $command := \langle \texttt{XMLHTTPREQUEST}, \text{URL}^{origin.\texttt{domain}}_{/\texttt{startLogin}}, \texttt{POST}, s'.\texttt{email}, s'.\texttt{refXHR}\rangle$
    let $s'.\texttt{q} :=$ `expectStartLoginResponse`
  case `expectStartLoginResponse`
    let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
    let $input :=$ CHOOSEFIRSTINPUTPAT$(scriptinputs, pattern)$
    if $input \not\equiv \bot$ **then**
      let $s'.\texttt{loginSessionToken} := \pi_2(input)[\texttt{loginSessionToken}]$
      let $s'.\texttt{tagKey} := \pi_2(input)[\texttt{tagKey}]$
      let $s'.\texttt{FWDDomain} := \pi_2(input)[\texttt{FWDDomain}]$
      let $loginURL := \pi_2(input)[\texttt{loginURL}]$
      let $command := \langle \texttt{HREF}, loginURL, \texttt{\_BLANK}, \langle\rangle\rangle$
      let $s'.\texttt{q} :=$ `expectFWDReady`
    end if
  case `expectFWDReady`
    let $fwdWindowNonce :=$ SUBWINDOWS$(tree,$ AUXDOCNONCE$(tree, docnonce))$.1.`nonce`
    let $pattern := \langle \texttt{POSTMESSAGE}, fwdWindowNonce, \langle s'.\texttt{FWDDomain}, \texttt{S}\rangle, \texttt{ready}\rangle$
    let $input :=$ CHOOSEFIRSTINPUTPAT$(scriptinputs, pattern)$
    if $input \not\equiv \bot$ **then**
      let $command := \langle \texttt{POSTMESSAGE}, fwdWindowNonce, \langle \texttt{tagKey}, tagKey\rangle, \langle s'.\texttt{FWDDomain}, \texttt{S}\rangle\rangle$
      let $s'.\texttt{q} :=$ `expectEIA`
    end if
  case `expectEIA`
    let $fwdWindowNonce :=$ SUBWINDOWS$(tree,$ AUXDOCNONCE$(tree, docnonce))$.1.`nonce`
    let $pattern := \langle \texttt{POSTMESSAGE}, fwdWindowNonce, \langle s'.\texttt{FWDDomain}, \texttt{S}\rangle, \langle \texttt{eia}, *\rangle\rangle$
    let $input :=$ CHOOSEFIRSTINPUTPAT$(scriptinputs, pattern)$
    if $input \not\equiv \bot$ **then**
      let $eia := \pi_2(\pi_4(input))$
      let $s'.\texttt{refXHR} := \lambda_1$
      let $body := \langle\langle \texttt{eia}, eia\rangle, \langle \texttt{loginSessionToken}, s'.\texttt{loginSessionToken}\rangle\rangle$
      let $command := \langle \texttt{XMLHTTPREQUEST}, \text{URL}^{origin.\texttt{domain}}_{/\texttt{login}}, \texttt{POST}, body, s'.\texttt{refXHR}\rangle$
      let $s'.\texttt{q} :=$ `expectServiceToken`
    end if
stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$

Algorithm F.6: Relation of *script_idp*.

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

1: **let** $s' := scriptstate$
2: **let** $command := \langle\rangle$
3: **let** $origin := \mathsf{GETORIGIN}(tree, docnonce)$
4: **switch** $s'.\mathtt{q}$ **do**
5: **case** `start`
6: **let** $email := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{email}]$
7: **let** $tag := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{tag}]$
8: **let** $FWDDomain := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{FWDDomain}]$
9: **let** $body := \langle\langle \mathtt{email}, email\rangle, \langle \mathtt{password}, secret\rangle, \langle \mathtt{tag}, tag\rangle, \langle \mathtt{FWDDomain}, FWDDomain\rangle\rangle$
10: **let** $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}_{/\mathtt{sign}}^{origin.\mathtt{domain}}, \mathtt{POST}, body, \bot\rangle$
11: **let** $s'.\mathtt{q} := \mathtt{expectIA}$
12: **case** `expectIA`
13: **let** $pattern := \langle \mathtt{XMLHTTPREQUEST}, *, *\rangle$
14: **let** $input := \mathsf{CHOOSEFIRSTINPUTPAT}(scriptinputs, pattern)$
15: **if** $input \not\equiv \bot$ **then**
16: **let** $iaKey := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{iaKey}]$
17: **let** $FWDDomain := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{FWDDomain}]$
18: **let** $tag := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{tag}]$
19: **let** $eia := \mathsf{enc_s}(\pi_2(input), iaKey)$
20: **let** $url := \langle \mathtt{URL}, \mathtt{S}, FWDDomain, /, \langle\langle \mathtt{tag}, tag\rangle, \langle \mathtt{eia}, eia\rangle\rangle\rangle$
21: **let** $command := \langle \mathtt{IFRAME}, url, \mathtt{\_SELF}\rangle$
22: **let** $s'.\mathtt{q} := \mathtt{stop}$
23: **end if**
24: **stop** $\langle s', cookies, localStorage, sessionStorage, command\rangle$

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

Definition 101. A *scriptstate s of script_idp* is a term of the form $\langle q, email\rangle$ with $q \in \mathbb{S}$, $email \in \mathsf{ID} \cup \{\langle\rangle\} \in \mathcal{T}$. We call the scriptstate s an *initial scriptstate* of *script_idp* iff $s \sim \langle \mathtt{start}, *\rangle$. ◇

We formally specify the relation *script_idp* of the LD's scripting process in Algorithm F.6.

Forwarder Script (script_fwd). This script models the contents of the forwarder iframe.

Definition 102. 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 $s \equiv \mathtt{start}$. ◇

We formally specify the relation *script_fwd* of the FWD's scripting process in Algorithm F.7.

Algorithm F.7: Relation of *script_fwd*.

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

1: **let** $s' := scriptstate$
2: **let** $command := \langle\rangle$
3: **let** $target :=$ OPENERWINDOW($tree$,PARENTDOCNONCE($tree, docnonce$))
→ Determine the opener of the parent window
4: **switch** $s'.\mathtt{q}$ **do**
5: **case** `start`
6: **let** $command := \langle$`POSTMESSAGE`, $target$, `ready`, $\bot\rangle$
→ Prepare a postMessage to the opener of the parent window to indicate that the forwarder script is ready.
7: **let** $s'.\mathtt{q} :=$ `expectTagKey` → Modify the scriptstate such that q is set to `expectTagKey`.
8: **case** `expectTagKey`
9: **let** $pattern := \langle$`POSTMESSAGE`$, target, *, \langle$`tagKey`$, *\rangle\rangle$
10: **let** $input :=$ CHOOSEFIRSTINPUTPAT($scriptinputs, pattern$)
→ Take the first input for the script that matches the pattern above (a postMessage that was sent from the parent's opener and contains a dictionary with the dictionary key `tagKey`).
11: **if** $input \not\equiv \bot$ **then**
12: **let** $tagKey := \pi_2(\pi_4(input))$ → Extract *tagKey* from the input.
13: **let** $tag :=$ GETPARAMETERS($tree, docnonce$)[`tag`]
→ Extract *tag* from the URL parameters of this script's document.
14: **let** $eia :=$ GETPARAMETERS($tree, docnonce$)[`eia`]
→ Extract *eia* from the URL parameters of this script's document.
15: **let** $rpOrigin := \langle \mathsf{dec_s}(tag, tagKey).1, \mathtt{S}\rangle$
→ Decrypt *tag* using *tagKey* to extract the RP's origin.
16: **let** $command := \langle$`POSTMESSAGE`$, target, \langle$`eia`$, eia\rangle, rpOrigin\rangle$
→ Prepare a postMessage that is sent to the parent's opener such that this receiving document must have the origin of RP and that contains the *eia*.
17: **let** $s'.\mathtt{q} :=$ `stop` → Modify the scriptstate such that q is set to `stop`.
18: **end if**
19: **stop** $\langle s', cookies, localStorage, sessionStorage, command\rangle$

F.1.11. Important Events

As described in Section 3.2, we define events that (if they occur in a certain processing step) specify important actions of our SSO protocol.

Definition 103 (Refinement of Definition 5 for SPRESSO: Start of an SSO Flow). Let $\mathcal{SWS}$ be an SPRESSO Web system. Let ρ be a run of $\mathcal{SWS}$. Let $Q \in \rho$ be a processing step, b a browser, and r an RP. We write $\mathsf{started}(Q,b,r)$ iff in Q, the browser b is triggered and selects to run the script of a document and this script is `script_rp` and the document is stored in b under a secure origin[1] of r and —when executing the script— Lines 6ff. of Algorithm F.5 are executed. ◇

Note that `script_rp` is implemented as a state machine with the current state stored in the subterm q of the script's state. The initial value for q is `start`.

Definition 104 (Refinement of Definition 6 for SPRESSO: Selection of an IdP). Let $\mathcal{SWS}$ be an SPRESSO Web system. Let ρ be a run of $\mathcal{SWS}$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, and i an IdP. We write $\mathsf{selectedIdP}(Q,b,i)$ iff $\mathsf{started}(Q,b,r)$ and in Line 6 of Algorithm F.5 some identity id' with $\mathsf{governor}(id') = i$ was selected. ◇

Definition 105 (Refinement of Definition 7 for SPRESSO: Selection of an Identity). Let $\mathcal{SWS}$ be an SPRESSO Web system. Let ρ be a run of $\mathcal{SWS}$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, i an IdP, and id an identity. We write $\mathsf{selectedID}(Q,b,i,id)$ iff $\mathsf{started}(Q,b,r)$ and in Line 6 of Algorithm F.5 the value id was selected. ◇

Definition 106 (Refinement of Definition 8 for SPRESSO: User is Logged in at RP). Let $\mathcal{SWS}$ be an SPRESSO Web system. Let ρ be a run of $\mathcal{SWS}$. Let $Q \in \rho$ be a processing step, b a browser, r an RP, id an identity, and n some term. We write $\mathsf{loggedIn}(Q,b,r,id,n)$ iff in Q, (1) b receives a HTTPS response with n as its body, (2) b processes this response such that b adds n to the scriptinputs of some document with the script `script_rp` under a secure origin of r as a response to an XHR in the state of b, and (3) in the state of r (before and after Q — note that the state of r cannot be altered during Q), for the subterm *serviceTokens*, we have $serviceTokens[n] \equiv id$. ◇

[1] As for BrowserID, we assume a secure origin of RP here as —similar to reality— we cannot have any integrity of scripts and their states under an insecure origin.

F.1.12. SPRESSO Web System for Privacy Analysis

As mentioned above, for privacy analysis, we here refine the definition of an SPRESSO Web system.

Definition 107 (SPRESSO Web System for Privacy Analysis). Let $\mathcal{SWS} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ be an SPRESSO Web system with $\mathcal{W}^{sp} = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$, $\mathsf{Hon} = \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP} \cup \mathsf{FWD} \cup \mathsf{DNS}$ (as described above), $\mathsf{RP} = \{r_1, r_2\}$, $\mathsf{FWD} = \{\mathsf{fwd}\}$, $\mathsf{DNS} = \{\mathsf{dns}\}$, r_1 and r_2 two (honest) relying parties, fwd an honest forwarder, dns an honest DNS server. Let $\mathsf{attacker} \in \mathsf{Web}$ be some Web attacker. Let dr be a domain of r_1 or r_2 and $b(dr)$ a challenge browser. Let $\mathsf{Hon}' := \{b(dr)\} \cup \mathsf{RP} \cup \mathsf{FWD} \cup \mathsf{DNS}$, $\mathsf{Web}' := \mathsf{Web}$, and $\mathsf{Net}' := \emptyset$ (i.e., there is no network attacker). Let $\mathcal{W}^{sp'} := \mathsf{Hon}' \cup \mathsf{Web}' \cup \mathsf{Net}'$. Let $\mathcal{S}' := \mathcal{S} \setminus \{\texttt{script\_idp}\}$ and script' be accordingly. We call $\mathcal{SWS}^{priv}(dr) = (\mathcal{W}^{sp'}, \mathcal{S}', \mathsf{script}', E^0, \mathsf{attacker})$ an *SPRESSO web system for privacy analysis* iff the domain `fwddomain` 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 `FWDdomain` is set to `fwddomain`. 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 $\mathcal{W}^{sp'}$ and in the initial states $s_0^{r_1}$, $s_0^{r_2}$ of both relying parties, the DNS address is $\mathsf{addr}(\mathsf{dns})$. 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 $\rho = ((S^0,E^0,N^0),\ldots,(S^i,E^i,N^i),\ldots,(S^{|\sigma|},E^{|\sigma|},N^{|\sigma|}))$ as all of its commands induce at most one processing step for the i-th configuration.

F.2. Proof of Theorem 5 (Security w.r.t. Authentication)

Before we prove Theorem 5, we show some general properties of the $\mathcal{SWS}^{auth}$.

Let $\mathcal{SWS}^{auth} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ be an SPRESSO Web system with a network attacker. In the following, we write $s_x = (S^x, E^x, N^x)$ for the states of a web system.

Definition 108. In a similar fashion to Definition 66, 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 F.3 was called). ◇

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

Lemma 17. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{SWS}^{auth}$ an honest relying party r (I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k\rangle, \mathsf{pub}(k'))$$

(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 $s_j \to s_{j+1}$ where the RP r leaks k to $\mathcal{W}^{sp} \setminus \{u, r\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W}^{sp} \setminus \{u, r\}$ 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 $\mathsf{pub}(k')$ in RP's keymapping $s_0.\texttt{keyMapping}$ (in its initial state).

4. If r accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and r is honest in s_j and u did not leak the symmetric key k to $\mathcal{W}^{sp} \setminus \{u, r\}$ prior to s_j, then either u or r created the HTTPS response m' to the HTTPS request m, in particular, 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 $\mathcal{W}^{sp} \setminus \{u, r\}$ in the processing step $s_j \to s_{j+1}$ 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_i. 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 F.3), a host header is always contained in HTTP requests by RPs. From Line 22 of Algorithm F.3 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 RP 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)). This proves (4). □

Lemma 18. For every honest relying party $r \in \mathsf{RP}$, every $s \in \rho$, every term $\langle host, wkDoc \rangle \in^{\langle\rangle} S(r).\texttt{wkCache}$ it holds that $wkDoc[\texttt{signkey}] \equiv \mathsf{pub}(\mathsf{signkey}(\mathsf{dom}^{-1}(host)))$ if $\mathsf{dom}^{-1}(host)$ is an honest IdP.

Proof. First, we can see that (in an honest RP) $S(r).\texttt{wkCache}$ can only be populated in Line 14 (of Algorithm F.3). There, the body of a received message m' is written to $S(r).\texttt{wkCache}$. 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 $request.\texttt{host}$ is the domain the original request was encrypted for and finally sent to.

With the condition of this lemma we see that $\mathsf{dom}^{-1}(request.\texttt{host})$ is an honest IdP, say, p. Lemma 17 applies here. As r emits HTTPS requests only for the well-known fixed path `/.well-known/spresso-info` and does not emit an HTTPS response for such an HTTPS request, we can see that p created the HTTPS response, and it was not altered by any other party. In Algorithm F.1 we can see that an honest IdP responds to requests to the well-known 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: $\langle\langle \texttt{signkey}, \mathsf{pub}(s'.\texttt{signkey})\rangle\rangle$. The term $s'.\texttt{signkey}$ is defined in Definition 97 to be $\mathsf{signkey}(p)$ and is never changed in Algorithm F.1.

Therefore, a pairing of the form $\langle request.\texttt{host}, x\rangle$ with

$$x[\texttt{signkey}] \equiv \mathsf{pub}(\mathsf{signkey}(\mathsf{dom}^{-1}(request.\texttt{host})))$$

is stored in $S(r).\texttt{wkCache}$. As this applies to all pairings in $S(r).\texttt{wkCache}$, this proves the lemma. □

Definition 109. 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 19. In a run ρ of $\mathcal{SWS}^{auth}$, for every state $s_j \in \rho$, every RP $r \in \mathsf{RP}$ that is honest in s_j, every $\langle n, i\rangle \in^{\langle\rangle} S^j(r).\texttt{serviceTokens}$, the following properties hold:

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

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

   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

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

   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:

$$\mathsf{enc_a}(\langle\langle \texttt{HTTPReq}, n_{\text{req}}, \texttt{POST}, d_r, /\texttt{login}, x, h, b\rangle, k\rangle, \mathsf{pub}(\mathsf{tlskey}(d_r)))$$

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

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

with

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

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

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

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

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

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

Proof. **(1).** The service token nonce n of service tokens $\langle n, i \rangle \in^{\langle\rangle} S^j(r).\texttt{serviceTokens}$ can only be contained in a response that is assembled in Lines 43ff of Algorithm F.3. The n is freshly chosen in Line 58, stored (along with the identity i) to $S^j(r).\texttt{serviceTokens}$ (actually to $S^q(r).\texttt{serviceTokens}$ for some $q \leq j$) in Line 60 and sent out in the service token response in Line 61f. The service tokens stored in $S^j(r).\texttt{serviceTokens}$ are not used or altered anywhere else. Therefore, each service token nonce is sent in exactly one (service token) response.
(2). From Line 43 of Algorithm F.3 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 43. The values of $b[\texttt{eia}]$, i, tag, and $iaKey$ are checked in Lines 52ff.
(5). In Line 55, the term ia is checked to be signed with the signature key stored in $S^q(r).\texttt{wkCache}$ indexed under the domain of the email address i (for some $q \leq j$). With Lemma 18, we can see that for the domain of the email address i this signature key is $\mathsf{signkey}(\mathsf{dom}^{-1}(i.\texttt{domain}))$. With $\mathsf{dom}^{-1}(i.\texttt{domain}) = \mathsf{governor}(i)$ 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.

□

We now show that Theorem 5 holds true. We want to show that every SPRESSO web system is secure w.r.t. authentication and therefore assume that there exists an SPRESSO web system that is not secure. We will lead this to a contradication and thereby show that all SPRESSO web systems are secure w.r.t. authentication.

In detail, we assume: *There exists an SPRESSO web system* $\mathcal{SWS}^{auth}$*, a run* ρ *of* $\mathcal{SWS}^{auth}$*, a state* $s_j = (S^j, E^j, N^j)$ *in* ρ*, a RP* $r \in \mathsf{RP}$ *that is honest in* S^j *with* $S^0(r).\texttt{FWDDomain}$ *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 $S^j(r).\texttt{serviceTokens}$ and derivable from the attackers knowledge in S^j (i.e., the term $\langle n,i\rangle$ is in $d_\emptyset(S^j(\mathsf{attacker}))$), and the browser b owning i is not fully corrupted and $\mathsf{governor}(i)$ 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 19 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 19 (4). Let $I := \mathsf{governor}(i)$. We know that I is an honest IdP. As such, it never leaks its signing key (see Algorithm F.1). Therefore, the signed subterm $ia := \mathsf{sig}(\langle tag, i, S(r).\texttt{FWDDomain}\rangle, \mathsf{signkey}(I))$ had to be created by the IdP I. An (honest) IdP creates signatures only in Line 25 of Algorithm F.1.

Lemma 20. 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 identitiy i is marked as being logged in, compare Line 22 of Algorithm F.1), then the request has to carry (in the request body) the password matching the identity i ($\mathsf{secretOfID}(i)$). 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)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 sent 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 $\mathcal{SWS}^{auth}$). □

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

Proof. First, $\langle d, \mathsf{S} \rangle$ for some $d \in \mathsf{dom}(I)$ is the only origin from which a script has access to the secret $\mathsf{secretOfID}(i)$ for the identity i (as defined in Appendix F.1.5).

With the general properties defined in Appendix B and the definition of Identity Providers in Appendix F.1.6, 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. □

Lemma 22. 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 TLS keys. It can therefore not decrypt the response if it becomes close-corrupted before receiving the response.

The remainder follows from the general properties defined in Appendix B. □

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 22). This would be a contradiction to the assumption (which requires that the IA arrived at the RP).

Lemma 23. 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 scriptstates 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 F.6) 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 F.6). 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 $S(r).\texttt{FWDDomain}$), we know that the domain to which the *ia* is forwarded is the same.

From our assumption, we know that $S(r).\texttt{FWDDomain}$ is the origin of an honest FWD in s_j. It is contacted over HTTPS, so the general properties defined in Appendix B apply. According to the definition of forwarders (Algorithm F.4), they only respond with *script_fwd*. The *ia* is therefore only forwarded to the FWD and its script *script_fwd*. □

Lemma 24. The script *script_fwd* forwards the *ia* only to the script *script_rp* loaded from the origin $\langle d_r, \texttt{S} \rangle$.

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 F.6) 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 F.4) 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. □

Lemma 25. 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 F.5, in particular Line 37 in conjunction with Line 3) and the fact that the RP document must have been loaded from the origin $\langle d_r, \texttt{S} \rangle$ (as shown above). □

With Lemmas 22–25 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 Appendix B) and it will not do anything with the contents of the response (see Algorithm F.5), in particular not leak it to the attacker or any corrupted party.

This is a contradication to the assumption, where we assumed that $\langle n,i\rangle \in d_{\emptyset}(S^j(\mathsf{attacker}))$. This shows every $\mathcal{SWS}^{auth}$ is secure w.r.t. authentication.

■

F.3. Proof of Theorem 6 (Security w.r.t. Session Integrity)

Let $\mathcal{SWS}^{auth} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0)$ be an SPRESSO Web system with a network attacker. In the following, we write $s_x = (S^x, E^x, N^x)$ for the states of a web system.

Let ρ be a run of $\mathcal{SWS}^{auth}$. Let b a browser, r an RP, id an identity, and n some term, such that $Q \in \rho$ be a processing step with $\mathsf{loggedIn}(Q,b,r,id,n)$, r being honest in Q_{login}, and b being honest in Q_{login}.

We know (by definition of $\mathsf{loggedIn}$) that n is indeed a service token for the identity id at r.

Applying Lemma 19 (1–4), we call the request corresponding to the service session $\langle n,id\rangle$ (the request that causes the RP to create a service session/service token) m and its response m', and (as in Lemma 19 (2)) we refer to the state of $\mathcal{SWS}^{auth}$ in the run ρ where r processes m by s_l.

Lemma 26. The request m was sent by *script_rp* loaded from the origin $\langle d_r, \mathtt{S}\rangle$ where d_r is some domain of r.

Proof. In Algorithm F.3, Line 44, 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 $\langle d_r, \mathtt{S}\rangle$, 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. Hence, m must have been sent by *script_rp* loaded from the origin $\langle d_r, \mathtt{S}\rangle$. □

Lemma 27. The request m contains a nonce *loginSessionToken* in its body such that

$$S^l(r).\texttt{loginSessions}[\mathit{loginSessionToken}].\texttt{email} \equiv id'$$

and b owns id', i.e., $\mathsf{ownerOfID}(id') = b$.

Proof. With Lemma 26 we know that the request was sent by *script_rp*. In Algorithm F.5 defining *script_rp*, in Line 36, 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 scriptstate (*loginSessionToken*). This part of the state is initially set to $\bot$ 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 scriptstate). 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 F.1.5). 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), ultimatively, the function *SENDSTARTLOGINRESPONSE* (Algorithm F.2) is called. There are two cases how this function can be called (see Line 36 of Algorithm F.3):

- 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.

After *SENDSTARTLOGINRESPONSE* is called, a new *loginSessionToken* is chosen and in the dictionary $S^x(r).\texttt{loginSessions}[\mathit{loginSessionToken}]$ 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 id' 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

$$S^l(r).\texttt{loginSessions}[\mathit{loginSessionToken}].\texttt{email} \equiv \mathit{id}' \ .$$

As shown above, due to the way id' is selected by the script, b owns id'. □

With Lemma 27, we can now show that $id = id'$: In Line 59 of Algorithm F.3, the service token is assembled. In particular, id is chosen to be $S^l(r).\texttt{loginSessions}[\mathit{loginSessionToken}].\texttt{email}$, and therefore $id = id'$ and b owns id. From the proof of Lemma 27 and the definition of *script_rp* we further know that the login session token must have been requested for id by *script_rp* in Line 8 in processing step, say Q_{start} and that *script_rp* only ever sends out such a request once (following the definition of the state machine of *script_rp*). Hence, we have that there exists $o \in \mathit{SSOSession}(\rho, b, r)$ with $\mathsf{started}(Q_{\text{start}}, b, r) \in o$, $\mathsf{selectedIdP}(Q_{\text{start}}, b, i)$, $\mathsf{selectedID}(Q, b, i, id)$, and $Q_{\text{login}} \in o$. ■

F.4. Proof of Theorem 7 (Privacy)

Here, we define an equivalence relation between configurations. Recall that for privacy analysis, we consider two SSO systems that are exactly the same up to the challenge domain set in one challenge browser (which is part of both systems). We show that for both systems there exists no processing step (induced by the same, but arbitrary schedule) which breaks (static) equivalence between the configurations of both SSO systems.

F.4.1. Definition of Equivalent Configurations

Let $\mathcal{SWS}_1^{priv} = (\mathcal{W}^{sp}{}_1, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ and $\mathcal{SWS}_2^{priv} = (\mathcal{W}^{sp}{}_2, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ 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 110 (Proto-Tags). We call a term of the form $\mathsf{enc_s}(\langle y, n\rangle, k)$ with the variable y as a placeholder for a domain, and n and k some nonces a *proto-tag*. ◇

Definition 111 (Term Equivalence up to Proto-Tags). Let $\theta = \{a_1, \ldots, a_l\}$ 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 $\tau \in \mathcal{T}_{\mathcal{N}}(\{x_1,\ldots,x_l\})$ such that $t_1 = (\tau[a_1/x_1,\ldots,a_l/x_l])[dr_1/y]$ and $t_2 = (\tau[a_1/x_1,\ldots,a_l/x_l])[dr_2/y]$. 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 $(d_1,\ldots,d_{|D|})$ and the elements in D' of the form $(d'_1,\ldots,d_{|D'|})$, we have that for all $i \in \{1,\ldots,|D|\}$: $d_i \rightleftharpoons_\theta d'_i$. We then write $D \rightleftharpoons_\theta D'$. ◇

Definition 112 (Equivalence of (and Invariants for) HTTP Requests). Let m_1 and m_2 be (potentially encrypted) HTTP requests and $\theta = \{a_1,\ldots,a_l\}$ 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 $g_j[\mathtt{eia}] \sim \mathsf{enc_s}(\mathsf{sig}(\langle \mathsf{enc_s}(\langle dr_j, * \rangle, *), *, \mathtt{fwddomain} \rangle, *), *)$ and the origin (HTTP header) of HTTP message in m_j is $\langle dr_j, \mathtt{S} \rangle$ 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 $g_1[\mathtt{loginSessionToken}] \equiv l'$, 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 $m_1 \doteq_\theta m_2$.

◇

Definition 113 (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 $t_1 \trianglerighteq_d^\theta (t_2, D)$. ◇

Definition 114 (Login Session Tokens for Proto Tags). 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 $l \in \mathsf{loginSessionTokens}(a, S_1, S_2)$ iff for any $i \in \{1,2\}$ and any $j \in \{1,2\}$ we have that $\pi_4(S_i(r_j).\mathtt{loginSessions}[l]) = a[dr_j/y]$. ◇

Definition 115 (Equivalence of (and Invariants for) States). Let θ be a set of proto-tags and H be a set of nonces. Let $K := \{k \mid \exists n : \mathsf{enc_s}(\langle y, n\rangle, k) \in \theta\}$. We call S_1 and S_2 γ-*equivalent under* (θ, H)iff the following conditions are met:

1. $S_1(\mathsf{fwd}) = S_2(\mathsf{fwd})$, and
2. $S_1(\mathsf{dns}) = S_2(\mathsf{dns})$, and
3. $S_1(\mathsf{r}_1)$ equals $S_2(\mathsf{r}_1)$ except for the subterms *pendingDNS*, *loginSessions* and *serviceTokens*, and
4. $S_1(\mathsf{r}_2)$ equals $S_2(\mathsf{r}_2)$ except for the subterms *pendingDNS*, *loginSessions* and *serviceTokens*, and
5. for two sets of terms D and D': $S_1(\mathsf{r}_1).\mathtt{loginSessions} \trianglerighteq^{\theta}_{dr_1} (S_2(\mathsf{r}_1).\mathtt{loginSessions}, D)$, $S_2(\mathsf{r}_2).\mathtt{loginSessions} \trianglerighteq^{\theta}_{dr_2} (S_1(\mathsf{r}_2).\mathtt{loginSessions}, D')$, and $D \rightleftharpoons_{\theta} D'$, and
6. for all entries x in the subterms `pendingDNS` of $S_1(\mathsf{r}_1)$, $S_1(\mathsf{r}_2)$, $S_2(\mathsf{r}_1)$, and $S_1(\mathsf{r}_2)$ it holds true that $\pi_2(x).\mathtt{host}$ is not a domain name known to the DNS server, and
7. the subterms `pendingRequest` of $S_1(\mathsf{r}_1)$, $S_1(\mathsf{r}_2)$, $S_2(\mathsf{r}_1)$, and $S_2(\mathsf{r}_2)$ are $\langle\rangle$, and
8. the subterm `wkCache` of $S_1(\mathsf{r}_1)$, $S_1(\mathsf{r}_2)$, $S_2(\mathsf{r}_1)$, and $S_2(\mathsf{r}_2)$ are equal and contain a public key for each domain registered in the DNS server, and
9. $\forall k \in K$: $k \notin d_{\emptyset}(\bigcup_{i\in\{1,2\},\, A\in \mathsf{Web}\cup\mathsf{Net}\cup\{\mathsf{dns},\mathsf{fwd}\}} S_i(A))$
10. for each attacker A: $S_1(A) \rightleftharpoons_{\theta} S_2(A)$, and
11. for all $a \in \theta$ and all attackers A we have that $\nexists\, l \in \mathsf{loginSessionTokens}(a, S_1, S_2)$ 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*, and *windows* and we have that
    a) $S_1(b_1).\mathtt{challenge} = dr_1 \wedge S_2(b_2).\mathtt{challenge} = dr_2$ or $S_1(b_1).\mathtt{challenge} = S_2(b_2).\mathtt{challenge} = \bot$, and
    b) $|S_1(b_1).\mathtt{pendingDNS}| = |S_2(b_2).\mathtt{pendingDNS}| =: j$, for all $i \in \{1, \ldots, j\}$, $q_1 := \pi_i(S_1(b_1).\mathtt{pendingDNS})$, $q_2 := \pi_i(S_2(b_2).\mathtt{pendingDNS})$ 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 term of the form $\langle \texttt{REQ}, x \rangle$ or a term of the form $\langle \texttt{XHR}, x, y \rangle$ with $x \in \mathcal{N}$ a nonce and $y \in \mathcal{N} \cup \{\bot\}$ a nonce or $\bot$, and

iv. if $\pi_2(v_1).\texttt{host} = dr_1 \wedge \pi_2(v_2).\texttt{host} = dr_2$,
then $\pi_2(v_1) \doteq_\theta \pi_2(v_2) \wedge \pi_2(v_1).\texttt{nonce} \in H$,
else $\pi_2(v_1) \rightleftharpoons_\theta \pi_2(v_2) \wedge \pi_2(v_1).\texttt{nonce} \notin H \wedge \nexists l \in L$ such that l is a subterm of $\pi_2(v_1)$,

and

c) $|S_1(b_1).\texttt{pendingRequests}| = |S_2(b_2).\texttt{pendingRequests}| =: j$, for all $i \in \{1, \ldots, j\}$, $v_1 := \pi_i(S_1(b_1).\texttt{pendingRequests})$, $v_2 := \pi_i(S_2(b_2).\texttt{pendingRequests})$ 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 term of the form $\langle \texttt{REQ}, x \rangle$ or a term of the form $\langle \texttt{XHR}, x, y \rangle$ with $x \in \mathcal{N}$ a nonce and $y \in \mathcal{N} \cup \{\bot\}$ a nonce or $\bot$, and

iv. if $\pi_2(v_1).\texttt{host} = dr_1 \wedge \pi_2(v_2).\texttt{host} = dr_2$,
then $\pi_2(v_1) \doteq_\theta \pi_2(v_2) \wedge \pi_2(v_1).\texttt{nonce} \in H \wedge \pi_4(v_1) \in \mathsf{addr}(r_1) \wedge \pi_4(v_2) \in \mathsf{addr}(r_2)$,
else $\pi_2(v_1) \rightleftharpoons_\theta \pi_2(v_2) \wedge \pi_2(v_1).\texttt{nonce} \notin H \wedge \pi_4(v_1) = \pi_4(v_2) \wedge \nexists l \in L$ such that l is a subterm of $\pi_2(v_1)$,

and

d) there is no $k \in K$ such that

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

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

e) $S_1(b_1).\texttt{windows}$ equals $S_2(b_2).\texttt{windows}$ with the exception of the subterms *location*, *referrer*, *scriptstate*, and *scriptinputs* of some document terms pointed to by $\mathsf{Docs}^+(S_1(b_1)) = \mathsf{Docs}^+(S_2(b_2)) =: J$. For all $j \in J$ we have that:

i. there is no $k \in K$ such that

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

ii. if $S_1(b_1).\overline{j}.\texttt{origin} \in \{\langle dr_1, \texttt{S}\rangle, \langle dr_2, \texttt{S}\rangle\}$ then

A. $S_1(b_1).\overline{j}.\texttt{script} \equiv \texttt{script\_rp}$, and

B. $S_1(b_1).\overline{j}.\texttt{headers}[\texttt{ReferrerPolicy}] \equiv \texttt{noreferrer}$, and

C. $S_1(b_1).\overline{j}.\texttt{location}$ and $S_2(b_2).\overline{j}.\texttt{location}$ 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

D. $S_1(b_1).\overline{j}.\texttt{referrer}$ and $S_2(b_2).\overline{j}.\texttt{referrer}$ are term-equivalent under θ, and

E. $S_1(b_1).\overline{j}.\texttt{scriptstate} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptstate}$ and if $\exists l \in L$ such that l is a subterm of $S_1(b_1).\overline{j}.\texttt{scriptstate}$, then $S_1(b_1).\overline{j}.\texttt{location.host} \equiv dr_1$ and $S_2(b_2).\overline{j}.\texttt{location.host} \equiv dr_2$, and

F. for $p \in \{$

$$\langle \texttt{XMLHTTPREQUEST}, *, * \rangle,$$
$$\langle \texttt{POSTMESSAGE}, *, \langle \texttt{fwddomain}, \texttt{S}\rangle, \texttt{ready}\rangle,$$
$$\langle \texttt{POSTMESSAGE}, *, \langle \texttt{fwddomain}, \texttt{S}\rangle, \langle \texttt{eia}, *\rangle\rangle$$

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

G. if $\exists l \in L$ such that l is a subterm of $S_1(b_1).\overline{j}.\texttt{scriptinputs}$, then $S_1(b_1).\overline{j}.\texttt{location.host} \equiv dr_1$ and $S_2(b_2).\overline{j}.\texttt{location.host} \equiv dr_2$, and

H. $\forall k \in K$: k is not contained in any subterm of $S_1(b_1).\overline{j}.\texttt{scriptstate}$ except for $S_1(b_1).\overline{j}.\texttt{scriptstate.tagKey}$, and

- $S_1(b_1).\overline{j}.\texttt{origin} \not\equiv \langle dr_1, \texttt{S}\rangle$
  $\implies k \not\equiv S_1(b_1).\overline{j}.\texttt{scriptstate.tagKey}$, and
- $S_1(b_1).\overline{j}.\texttt{origin} \not\equiv \langle dr_1, \texttt{S}\rangle$
  $\implies k \notin d_\emptyset(S_1(b_1).\overline{j}.\texttt{scriptinputs})$, and
- $S_2(b_2).\overline{j}.\texttt{origin} \not\equiv \langle dr_2, \texttt{S}\rangle$
  $\implies k \not\equiv S_2(b_2).\overline{j}.\texttt{scriptstate.tagKey}$, and

- $S_2(b_2).\overline{j}.\texttt{origin} \not\equiv \langle dr_2, \texttt{S}\rangle$
  $\implies k \notin d_\emptyset(S_2(b_2).\overline{j}.\texttt{scriptinputs})$, and

iii. if $S_1(b_1).\overline{j}.\texttt{origin} = \langle \texttt{fwddomain}, \texttt{S}\rangle$ then

A. $S_1(b_1).\overline{j}.\texttt{script} \equiv \texttt{script\_fwd}$, and

B. $S_1(b_1).\overline{j}.\texttt{location} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{location}$, and

C. $S_1(b_1).\overline{j}.\texttt{scriptstate} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptstate}$, and

D. for $p = \langle \texttt{POSTMESSAGE}, *, *, \langle \texttt{tagKey}, *\rangle\rangle$, $x_1 = S_1(b_1).\overline{j}.\texttt{scriptinputs}|\, p$, and $x_2 = S_2(b_2).\overline{j}.\texttt{scriptinputs}|\, p$ we have that for all $i \in \{1, \ldots, |x|\}$:

- $\pi_2(\pi_i(x_1)) \rightleftharpoons_\theta \pi_2(\pi_i(x_2))$, and
- $\pi_1(\pi_3(\pi_i(x_1))) \rightleftharpoons_\theta \pi_1(\pi_3(\pi_i(x_2)))$ or
  $\pi_1(\pi_3(\pi_i(x_1))) = dr_1 \wedge \pi_1(\pi_3(\pi_i(x_2))) = dr_2$, and
- $\pi_2(\pi_3(\pi_i(x_1))) \rightleftharpoons_\theta \pi_2(\pi_3(\pi_i(x_2)))$, and
- $\pi_4(\pi_i(x_1)) \rightleftharpoons_\theta \pi_4(\pi_i(x_2))$, and

iv. if $S_1(b_1).\overline{j}.\texttt{origin} \notin \{\langle dr_1, \texttt{S}\rangle, \langle dr_2, \texttt{S}\rangle, \langle \texttt{fwddomain}, \texttt{S}\rangle\}$ then

A. $S_1(b_1).\overline{j}.\texttt{location} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{location}$, and

B. $S_1(b_1).\overline{j}.\texttt{referrer} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{referrer}$, and

C. $S_1(b_1).\overline{j}.\texttt{scriptstate} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptstate}$, and

D. $S_1(b_1).\overline{j}.\texttt{scriptinputs} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptinputs}$, and

E. there is no $k \in K$ such that

$$k \in d_{\mathcal{N}\setminus\{k\}}(\{S_1(b_1).\overline{j}.\texttt{scriptstate}, S_1(b_1).\overline{j}.\texttt{scriptinputs}\}), \text{ and}$$

F. $\nexists l \in L$ such that l is a subterm of $S_1(b_1).\overline{j}.\texttt{scriptstate}$ or of $S_1(b_1).\overline{j}.\texttt{scriptinputs}$, and

f) for $x \in \{\texttt{cookies}, \texttt{localStorage}, \texttt{sessionStorage}, \texttt{sts}\}$ we have that

$$S_1(b_1).x \rightleftharpoons_\theta S_2(b_2).x\ .$$

For the domains dr_1 and dr_2 there are no entries in the subterms x.

$\diamond$

Definition 116 (Equivalence of (and Invariants for) Events). Let θ be a set of proto-tags, L be a set of login session tokens, H be a set of nonces, and $K := \{k \mid \exists n : \mathsf{enc_s}(\langle y, n\rangle, k) \in \theta\}$. 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) $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$ 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, $m_1 \mathrel{\hat{=}}_\theta m_2$, and both requests are unencrypted or encrypted (i.e., m_1 and m_2 are the content of the encryption) and $m_1.\texttt{nonce} \in H$, 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 $e_i^{(1)}$) are the same except for the HTTP body $g_1 := m_1.\texttt{body}$ and the HTTP body $g_2 := m_2.\texttt{body}$ which have to be $g_1 \rightleftharpoons_\theta g_2$ and $m_1.\texttt{nonce} \in H$ and if g_1 contains a dictionary key `loginSessionToken` then there exists an $l' \in L$ such that

$$g_1[\texttt{loginSessionToken}] \equiv l' \,.$$

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 $e_i^{q(2)}$ is an HTTP(S) response from r_2 to b_2 and the bodies of both HTTP messages are of the form $\langle\langle \texttt{tagKey}, k\rangle, *, *, *\rangle$.

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 contains a `ReferrerPolicy` header with value `noreferrer`, but 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 `script_rp`.
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 117 (Equivalence of (and Invariants for) 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 $L := \bigcup_{a \in \theta} \mathsf{loginSessionTokens}(a, S_1, S_2)$, and $N_1 = N_2$. ◇

F.4.2. Privacy Proof

We here prove Theorem 7.

Let $\mathcal{SWS}^{priv} = (\mathcal{W}^{sp}, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ be an SPRESSO Web system for privacy analysis. To prove Theorem 7, we have to show that the SPRESSO Web systems $\mathcal{SWS}_1^{priv}$ and $\mathcal{SWS}_2^{priv}$ are indistinguishable (according to Definition 17), where $\mathcal{SWS}_1^{priv}$ and $\mathcal{SWS}_2^{priv}$ are defined as follows: Let $\mathcal{SWS}_1^{priv} = (\mathcal{W}^{sp}{}_1, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ and $\mathcal{SWS}_2^{priv} = (\mathcal{W}^{sp}{}_2, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ with $b_1 := b(dr_1) \in \mathcal{W}^{sp}{}_1$ and $b_2 := b(dr_2) \in \mathcal{W}^{sp}{}_2$ challenge browsers. Further, we require $\mathcal{W}^{sp} \setminus \{b\} = \mathcal{W}^{sp}{}_1 \setminus \{b_1\} = \mathcal{W}^{sp}{}_2 \setminus \{b_2\}$. We denote the following processes in $\mathcal{SWS}^{priv}$:

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 15, 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 $\sigma = (\zeta_1, \zeta_2, \ldots)$, 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 28. Let (S_1, E_1, N_1) and (S_2, E_2, N_2) be two α-equivalent configurations. Then we have that $S_1(\mathsf{attacker}) \approx S_2(\mathsf{attacker})$.

Proof. From the α-equivalence of (S_1, E_1, N_1) and (S_2, E_2, N_2) it follows that $S_1(\mathsf{attacker}) \rightleftharpoons_\theta S_2(\mathsf{attacker})$. From Condition 9 for γ-equivalence it follows that $\{k \mid \exists n : \mathsf{enc_s}(\langle y, n\rangle, k) \in \theta\} \cap d_\emptyset(\bigcup_{i\in\{1,2\},\, A\in \mathsf{Web}\cup\mathsf{Net}} S_i(A))$ (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. □

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 29, 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 30, show that for each configuration induced by a processing step in ζ, α-equivalence still holds true.

Lemma 29. 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 $L := \bigcup_{a\in\theta} \mathsf{loginSessionTokens}(a, S_1, S_2)$, and $N_1^0 = N_2^0 = N^0$.

Let $\theta = H = L = \emptyset$. Obviously, both latter conditions are true. For all parties $p \in \mathcal{W}^{sp}{}_1 \setminus \{b_1\}$, 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 115 are fulfilled. Hence, the initial configurations are α-equivalent. □

Lemma 30. 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 $\zeta = \langle ci, cp, \tau_{\text{process}}, cmd_{\text{switch}}, cmd_{\text{window}}, \tau_{\text{script}}, url\rangle$ 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 $L := \bigcup_{a\in\theta} \mathsf{loginSessionTokens}(a, S_1, S_2)$, $K := \{k \mid \exists n : \mathsf{enc_s}(\langle y, n\rangle, k) \in \theta\}$.

To induce a processing step, the ci-th message from E_1 or E_2, respectively, is selected. Following Definition 116, 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 2: 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

$$(S_1, E_1, N_1) \xrightarrow[p_1 \to E_{\text{out}}^{(1)}]{\langle a_1, f_1, m_1 \rangle \to p_1} (S_1', E_1', N_1') \text{ and}$$
$$(S_2, E_2, N_2) \xrightarrow[p_2 \to E_{\text{out}}^{(2)}]{\langle a_2, f_2, m_2 \rangle \to p_2} (S_2', E_2', N_2') \ .$$

Case $p_1 = \mathsf{fwd}$:. We know that one of the cases of Case 1 of Definition 116 must apply for $e_i^{(1)}$ and $e_i^{(2)}$. Out of these cases only Case 1a applies. Hence, $p_2 = \mathsf{fwd}$.

In the forwarder relation (Algorithm F.4), either Lines 3f. are executed in both processing steps or in none. It is easy to see that $E_{\text{out}}^{(1)} \rightleftharpoons_\theta E_{\text{out}}^{(2)}$ (containing at most one event). For this new event all cases of Definition 116 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 116 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 $p_1 = \mathsf{dns}$:. In this case, only Cases 1a, 1b and 1c of Definition 116 can apply. Hence, $p_2 = \mathsf{dns}$. We note that (*) applies analogously in all cases.

In the first case, it is easy to see that $E_{\text{out}}^{(1)} \rightleftharpoons_\theta E_{\text{out}}^{(2)}$. 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 116 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 $p_1 = \mathsf{r}_1$:. 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 $e^{(1)}$ is a DNS response, we know that $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$, which implies $p_2 = \mathsf{r}_1$. 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 115 (RPs only query domains unknown to dns) and both $e_i^{(1)}$ and $e_i^{(2)}$ fulfill Condition 6 of Definition 116 (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 $e^{(1)}$ is an HTTP response, we know that $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$, which implies $p_2 = \mathsf{r}_1$. From Condition 7 of Definition 115, 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 $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$, which implies $p_2 = \mathsf{r}_1$. 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 three possible types of HTTP requests that are accepted by r_1 in Algorithm F.3:

- *path* = / (index page), Line 28,
- *path* = /`startLogin` (start a login), Line 31, and
- *path* = /`login` (login), Line 43.

From the cases in Definition 116, 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 116, Case 1a: $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$. This case implies $p_2 = \mathsf{r}_1 = p_1$. As we see below, for the output events $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ (if any) only Case 1a of Definition 116 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. $E_{\text{out}}^{(1)} = E_{\text{out}}^{(2)}$, and Condition 5 of Definition 116 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 $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ contain a request matching Case 1a of Definition 116. As $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ 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 116 applies for the input events, i.e., they do not contain a subterm $l \in L$, the same 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 115 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 the function 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 $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$. The first element of the response's body is not a string and therefore Condition 5 of Definition 116 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* $= /$`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 44 of Algorithm F.3 and the second in Line 55. 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 $m_1 \rightleftharpoons_\theta m_2$.

  As we have that $m_1 \rightleftharpoons_\theta m_2$, and in particular $eia_1 := body_1[\texttt{eia}] \rightleftharpoons_\theta body_2[\texttt{eia}] =: eia_2$,

both, eia_1 and eia_2 have the same structure. If this structure does not match the expected structure (see Line 52f.), 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

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

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 116, 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 $p_2 = \mathsf{r}_2$.

We note that Condition 5 of Definition 116 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 116 is fulfilled.

As we see below, for the output events $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ (if any) only Case 1f of Definition 116 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 116, 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, $N_1 = N_1' = N_2 = N_2'$.

- $path = /\texttt{startLogin}$. As above, both email addresses in the input events are either equivalent and their domain (say, *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

$$m'_1 = \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, g_1\rangle, k)$$
$$m'_2 = \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, g_2\rangle, k)$$

with

$$g_1 = \langle\langle \texttt{tagKey}, v_2\rangle, \langle \texttt{loginSessionToken}, v_4\rangle, \langle \texttt{FWDDomain}, S_1(r_1).\texttt{FWDDomain}\rangle, \langle \texttt{loginURL}, \mathit{loginURL}_1\rangle\rangle$$
$$g_2 = \langle\langle \texttt{tagKey}, v_2\rangle, \langle \texttt{loginSessionToken}, v_4\rangle, \langle \texttt{FWDDomain}, S_2(r_2).\texttt{FWDDomain}\rangle, \langle \texttt{loginURL}, \mathit{loginURL}_2\rangle\rangle$$

with

$$\mathit{loginURL}_1 = \langle \texttt{URL}, \texttt{S}, \mathit{domain}, /.\texttt{well-known}/\texttt{spresso-login}, \mathit{params}_1\rangle$$
$$\mathit{loginURL}_2 = \langle \texttt{URL}, \texttt{S}, \mathit{domain}, /.\texttt{well-known}/\texttt{spresso-login}, \mathit{params}_2\rangle$$

with

$$\mathit{params}_1 = \langle\langle \texttt{email}, \mathit{email}\rangle, \langle \texttt{tag}, \mathit{tag}_1\rangle, \langle \texttt{iaKey}, v_3\rangle, \langle \texttt{FWDDomain}, S_1(r_1).\texttt{FWDDomain}\rangle\rangle$$
$$\mathit{params}_2 = \langle\langle \texttt{email}, \mathit{email}\rangle, \langle \texttt{tag}, \mathit{tag}_2\rangle, \langle \texttt{iaKey}, v_3\rangle, \langle \texttt{FWDDomain}, S_2(r_2).\texttt{FWDDomain}\rangle\rangle$$

with

$$\mathit{tag}_1 = \mathsf{enc_s}(\langle \mathit{dr}_1, v_1\rangle, v_2)$$
$$\mathit{tag}_2 = \mathsf{enc_s}(\langle \mathit{dr}_2, v_1\rangle, v_2)$$

Obviously, m'_1 equals m'_2 up to tag_1 and tag_2. For $N_1 = N_2 = (n_1, n_2, \dots)$, we set $\theta' = \theta \cup \{\mathsf{enc_s}(\langle y, n_1\rangle, n_2)\}$, $N'_1 = N'_2 = (n_5, \dots)$ (as exactly four nonces are chosen in both processing steps), and $L' = L \cup \{n_4\}$. We have that $\mathit{tag}_1 \rightleftharpoons_{\theta'} \mathit{tag}_2$ and hence, $m'_1 \rightleftharpoons_{\mathit{theta}'} m'_2$. The receiver of both messages is the browser b_1 or b_2, respectively. Obviously, it holds that $L' = \bigcup_{a \in \theta'} \mathsf{loginSessionTokens}(a, S'_1, S'_2)$ and there exists an $l' \in L'$ such that $g_1[\texttt{loginSessionToken}] \equiv l'$. As Conditions 1f and 3 of Definition 116 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 $N_1' = N_2'$.)

- $path = /\texttt{login}$.

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

  (A) First, there are two additional checks, the first in Line 44 of Algorithm F.3 checks the origin header and the second in Line 55 checks the identity assertion.

  As we know that $m_1 \doteq_\theta m_2$, 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

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

  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 $p_1 = \mathsf{r}_2$:. This case is analogue to the case $p_1 = \mathsf{r}_1$ above. Note that the Case 1e of Definition 116 cannot occur by definition.

Case $p_1 = \mathsf{b}_1$:. $\implies p_2 = \mathsf{b}_2$

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 116 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 115, 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 67 of Algorithm A.9. Further, we note that the resolved address contained in the DNS response has to be an IP address.

From Condition 12b of Definition 115, 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 $E^{(1)}_{\text{out}}$ and $E^{(2)}_{\text{out}}$, we have that in Case (I) $E^{(1)}_{\text{out}} \rightleftharpoons_\theta E^{(2)}_{\text{out}}$ and hence Condition 1a of Definition 116 is fulfilled. In Case (II), the output messages fulfill Condition 1e.

From Condition 2 of Definition 116, we know that no $l \in L$ is contained in the DNS responses. Further, we know from Condition 1d of Definition 116 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 115, 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 115 holds true.

We also have that no $k \in K$ is contained in the response (with Condition 3 of Definition 115). 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 115) 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 reference for a "normal" HTTP(S) request (the reference is a term of the form $\langle \texttt{REQ}, x \rangle$ with x being a nonce used as a window reference), or (II) the reference is a reference for an XHR (the reference is a term of the form $\langle \texttt{XHR}, x, y \rangle$ with x being a nonce used as a window reference and y some term chosen by the script that sent the request). From Condition 12c of Definition 115 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 116 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 115 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 116).

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

$$\langle v_7, \mathit{location}, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \langle\rangle, \langle\rangle, \top \rangle$$

with

$$\mathit{location} = \langle \mathtt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters} \rangle \ .$$

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 115. 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 115 is satisfied.

As we have that both incoming messages were encrypted messages (see Case 7 of Definition 116) and both messages come from r_1 and r_2, respectively, and therefore *script* is `script_rp` (see Case 5 of Definition 116) we have to check

Condition 12(e)ii of Definition 115 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 term-equivalent up to θ in both cases (note that for dr_1 and dr_2, we have that their Referrer Policies always prevent this header to be set, see Case 12(e)iiB of Definition 115).

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 $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$ (Case 1a of Definition 116), 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 $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$ 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 115). 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 115). 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 tupel, say, $\mathit{reference} = \langle \texttt{XHR}, \mathit{docnonce}, \mathit{xhrref} \rangle$, 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 116 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 115 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 116).

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 $\langle \texttt{XMLHTTPREQUEST}, g_j.\texttt{body}, xhrref \rangle$, where g_j is the HTTP body of m_j. With $g_1 \rightleftharpoons_\theta g_2$ and $xhrref \in \mathcal{N} \cup \{\bot\}$, 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 $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$ (Case 1a of Definition 116), 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 $\langle \texttt{XMLHTTPREQUEST}, g_j.\texttt{body}, xhrref \rangle$, where g_j is the HTTP body of m_j. With $e_i^{(1)} \rightleftharpoons_\theta e_i^{(2)}$ (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}.

`script` **(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 the window exists in b_2 and that $S_1(b_1).\bar{j}.\texttt{script} \equiv S_2(b_2).\bar{j}.\texttt{script}$. We now distinguish between the following cases, which cover all possible states of the windows/documents:

1. $S_1(b_1).\bar{j}.\texttt{origin} \in \{\langle dr_1, \texttt{S}\rangle, \langle dr_2, \texttt{S}\rangle\}$.

   In this case, it immediately follows from Case 12(e)iiA of Definition 115 that $S_1(b_1).\bar{j}.\texttt{script} \equiv \texttt{script\_rp}$.

   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 $S_1(b_1).\bar{j}.\texttt{scriptstate.q} = S_2(b_2).\bar{j}.\texttt{scriptstate.q}$ or the scriptstate 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 the states of the script $S_1(b_1).\bar{j}.\texttt{scriptstate.q} = S_2(b_2).\bar{j}.\texttt{scriptstate.q}$:

   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 $\langle dr_1, \texttt{S}\rangle$ or $\langle dr_2, \texttt{S}\rangle$ or it is (b) $\langle dr_1, \texttt{S}\rangle$ in b_1 and $\langle dr_2, \texttt{S}\rangle$ 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 scriptstate, 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 116. 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)ii of Definition 115 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.

The script always outputs the same command to the browser: it commands to navigate a new window to a URL that is taken as an unaltered subterm from *scriptinput*. Hence, the URLs used in both cases are also term-equivalent up to θ.

We have to distinguish between to cases: Either the origin of the document is (i) equal in b_1 and b_2, or (ii) the origin is $\langle dr_1, \mathtt{S} \rangle$ in b_1 and $\langle dr_2, \mathtt{S} \rangle$ 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 URL that the script outputs. In the second case (ii), however, we have that such a subterm may be contained in *scriptinput*.

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 116. In any case, 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 `CHALLENGE`[2] and δ-equivalent otherwise.

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 116 is fulfilled, or the output messages fulfill Condition 1e.

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 the scriptstates are term-equivalent up to θ, i.e., $S_1(b_1).\overline{j}.\texttt{scriptstate} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptstate}$, 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 $\langle \texttt{fwddomain}, \texttt{S} \rangle$ 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 the scriptstates are term-equivalent up to θ, i.e., $S_1(b_1).\overline{j}.\texttt{scriptstate} \rightleftharpoons_\theta S_2(b_2).\overline{j}.\texttt{scriptstate}$, 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.

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

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 $\langle dr_1, \mathtt{S}\rangle$ or $\langle midr_2, \mathtt{S}\rangle$ or it is (b) $\langle dr_1, \mathtt{S}\rangle$ in b_1 and $\langle midr_2, \mathtt{S}\rangle$ 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

$$g_j[\mathtt{eia}] \sim \mathsf{enc_s}(\mathsf{sig}(\langle \mathsf{enc_s}(\langle dr_j, * \rangle, *), *, \mathtt{fwddomain}\rangle, *), *)\,.$$

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 116. 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. $S_1(b_1).\overline{j}.\mathtt{origin} = \langle \mathtt{fwddomain}, \mathtt{S}\rangle$.

   It immediately follows that $S_1(b_1).\overline{j}.\mathtt{script} \equiv \mathtt{script\_fwd}$ in this case.

   As above, we have that either the scriptstates are term-equivalent up to θ, i.e., $S_1(b_1).\overline{j}.\mathtt{scriptstate.q} = S_2(b_2).\overline{j}.\mathtt{scriptstate.q}$, or the scriptstate 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 the states of the script $S_1(b_1).\bar{j}.\texttt{scriptstate.q} = S_2(b_2).\bar{j}.\texttt{scriptstate.q}$:

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)iii of Definition 115 we know that in *scriptinputs* either such a postMessage exists in both browsers or in none.

As the *tag* contained in the postMessages is term-equivalent under proto-tags *theta*, 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 *theta*. 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.

3. $S_1(b_1).\bar{j}.\texttt{origin} \notin \{\langle dr_1, \texttt{S}\rangle, \langle dr_2, \texttt{S}\rangle, \langle \texttt{fwddomain}, \texttt{S}\rangle\}$.

Here, we assumte that the script in this case is the 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 115) 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) $\langle \texttt{HREF}, url, hrefwindow, noreferrer \rangle$: Here, the browser calls the function 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 115. A DNS request is created in accordance with Condition 1b or Condition 1a of Definition 116. The same number of nonce is chosen in both processing steps, and therefore α-equivalence holds.

c) $\langle \texttt{IFRAME}, url, window \rangle$ This case is completely parallel to Case 3b.

d) $\langle \texttt{FORM}, url, method, data, hrefwindow \rangle$ This case is parallel to Case 3b, except that an Origin header is added. Its properties are the same as those of the Referer header in Case 3b.

e) $\langle \texttt{SETSCRIPT}, window, script \rangle$ 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) $\langle \texttt{SETSCRIPTSTATE}, window, scriptstate \rangle$ This case is parallel to Case 3e.

g) $\langle \texttt{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$ This case is parallel to Case 3b with the addition of the Origin header (see Case 3d) 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) $\langle \texttt{BACK}, window \rangle$, $\langle \texttt{FORWARD}, window \rangle$, and $\langle \texttt{CLOSE}, window \rangle$ 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) $\langle \texttt{POSTMESSAGE}, window, message, origin \rangle$ In this case, a term containing *message* (term-equivalent under θ) is added to a document's `scriptinput` term. If the *origin* is $\bot$, 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 $\langle dr_1, \texttt{S} \rangle$ 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 $\langle \texttt{fwddomain}, \texttt{S} \rangle$.

`urlbar` **(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 115 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 116.

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').

`reload` **(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.

`forward` **(navigate window forward) and** `back` **(navigate window backward):** In both cases, the state of b_1 and b_2 are modified in exactly the same way, i.e., only the information about which document is active in some window is modified. Therefore, we have α-equivalence for (S_1', E_1', N_1') and (S_2', E_2', N_2').

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 116 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 115 we have that $S_1(A) \rightleftharpoons_\theta S_2(A)$ and with Case 9 and Case 3 of Definition 116 it follows immediately that the attacker cannot decrypt any of the tags in θ in its knowledge. Further, in the attackers 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 115), 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 7. ■

Bibliography

[AF01] Martín Abadi and Cédric Fournet. „Mobile Values, New Names, and Secure Communication“. In: *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001)*. ACM Press, 2001, pp. 104–115.

[Adi11] Ben Adida. *How BrowserID differs from OpenID*. Identity at Mozilla Blog. July 15, 2011. URL: http://identity.mozilla.com/post/7669886219/how-browserid-differs-from-openid. Archived version available at https://web.archive.org/web/20170410145251/http://identity.mozilla.com/post/7669886219/how-browserid-differs-from-openid.

[Adi+13] Ben Adida et al. *BrowserID Specification*. BrowserID Specification Repository. Feb. 26, 2013. URL: https://github.com/mozilla/id-specs.

[Akh+10] Devdatta Akhawe, Adam Barth, Peifung Eric Lam, John Mitchell, and Dawn Song. „Towards a Formal Foundation of Web Security“. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*. IEEE Computer Society, 2010, pp. 290–304.

[Akh+16] Devdatta Akhawe, Frederik Braun, François Marier, and Joel Weinberger, eds. *Subresource Integrity – W3C Recommendation 23 June 2016*. June 23, 2016. URL: https://www.w3.org/TR/SRI/.

[Akh+14] Devdatta Akhawe, Frederik Braun, Joel Weinberger, and Mike West, eds. *Subresource Integrity – W3C First Public Working Draft 18 March 2014*. Mar. 18, 2014. URL: https://www.w3.org/TR/2014/WD-SRI-20140318/.

[Ann14] Anne van Kesteren, ed. *Cross-Origin Resource Sharing, W3C Recommendation*. Jan. 16, 2014. URL: http://www.w3.org/TR/2014/REC-cors-20140116/.

[Arm+13] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti. „An authentication flaw in browser-based Single Sign-On protocols: Impact and remediations“. In: *Computers & Security* 33 (2013), pp. 41–58.

[Arm+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and Llanos Tobarra. „Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps“. In: *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*. Ed. by Vitaly Shmatikov. ACM, 2008, pp. 1–10.

[Bai+13] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song 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.

[Bam+18a] Will Bamberg et al. *Persona FAQ*. Mozilla Developer Network Wiki. Sept. 20, 2018. URL: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona/FAQ.

[Bam+18b] Will Bamberg et al. *Why Persona for Mozilla?* Mozilla Developer Network Wiki. Sept. 20, 2018. URL: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona/Why_Persona.

[Ban+13] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. „Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage“. In: *Principles of Security and Trust - Second International Conference, POST 2013*. Ed. by David A. Basin and John C. Mitchell. Vol. 7796. Lecture Notes in Computer Science. Springer, 2013, pp. 126–146.

[Ban+14] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. „Discovering Concrete Attacks on Website Authorization by Formal Analysis“. In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[BNR06] Melissa Bateson, Daniel Nettle, and Gilbert Roberts. „Cues of being watched enhance cooperation in a real-world setting“. In: *Biology letters* 2.3 (2006), pp. 412–414.

[Ber+14] Robin Berjon et al., eds. *HTML5, W3C Recommendation*. Oct. 28, 2014. URL: http://www.w3.org/TR/html5/.

[Ber89] Timothy J. Berners-Lee. *Information management: A proposal*. Technical Report. Mar. 1989. URL: https://www.w3.org/History/1989/proposal.html.

[Bha+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. „Everest: Towards a Verified, Drop-in Replacement of HTTPS“. In: *2nd Summit on Advances in Programming Languages*. May 2017.

[Bla13] Bruno Blanchet. „Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif“. In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Vol. 8604. Lecture Notes in Computer Science. Springer, 2013, pp. 54–87.

[BP10] Aaron Bohannon and Benjamin C. Pierce. „Featherweight Firefox: formalizing the core of a web browser“. In: *Proceedings of the 2010 USENIX conference on Web application development*. USENIX Association, 2010, pp. 11–11.

[BCG12] Egon Börger, Antonio Cisternino, and Vincenzo Gervasi. „Contribution to a Rigorous Analysis of Web Application Frameworks“. In: *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012*. Ed. by John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene. Vol. 7321. Lecture Notes in Computer Science. Springer, 2012, pp. 1–20.

[Bug13a] Bugzilla@Mozilla. *Bug 868967 - Attacker is able to inject own assertion in login process*. May 2013. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=868967.

[Bug13b] Bugzilla@Mozilla. *Bug 920030 - Identity Forgery in BrowserID-Sideshow*. Sept. 2013. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=920030.

[Bug13c] Bugzilla@Mozilla. *Bug 920301 - Identity Forgery in BrowserID-BigTent*. Sept. 2013. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=920301.

[Bug14] Bugzilla@Mozilla. *Bug 1064254 - Identity Injection Attack on Persona by Malicious IdP*. Sept. 2014. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1064254. Access restricted.

[Bug15] Bugzilla@Mozilla. *Bug 1120255 - Privacy leak in Persona*. Jan. 2015. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1120255. Access restricted.

[BW17] Adam Barth and Mike West. *Cookies: HTTP State Management Mechanism*. Internet-Draft draft-ietf-httpbis-rfc6265bis-02. Work in progress. Internet Engineering Task Force, Aug. 7, 2017. URL: https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02.

[Cer+08] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, and Christopher Walstad. „Breaking and Fixing Public-key Kerberos“. In: *Inf. Comput.* 206.2-4 (2008), pp. 402–424.

[CCD11] Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. „Trace equivalence decision: negative tests and non-determinism“. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011, pp. 321–330.

[DW14] Michael Dietz and Dan S. Wallach. „Hardening Persona – Improving Federated Web Login“. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[DY83] Danny Dolev and Andrew C. Yao. „On the Security of Public-Key Protocols“. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[ES17] Jochen Eisinger and Emily Stark, eds. *Referrer Policy, Editor's Draft*. Apr. 20, 2017. URL: https://w3c.github.io/webappsec-referrer-policy/.

[F*] *F*: A Higher-Order Effectful Language Designed for Program Verification*. Accessed: 2019-07-05. URL: https://www.fstar-lang.org.

[Fau+17] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon, eds. *HTML 5.2, W3C Recommendation*. Dec. 14, 2017. URL: http://www.w3.org/TR/html52/.

[Fet18] Daniel Fett. „An Expressive Formal Model of the Web Infrastructure“. PhD thesis. University of Stuttgart, Germany, 2018.

[FHK19] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. „An Extensive Formal Security Analysis of the OpenID Financial-grade API“. In: *40th IEEE Symposium on Security and Privacy (S&P 2019)*. To appear. IEEE Computer Society, 2019.

[FKS14a] Daniel Fett, Ralf Küsters, and Guido 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)*. IEEE Computer Society, 2014, pp. 673–688.

[FKS14b] Daniel Fett, Ralf Küsters, and Guido Schmitz. *An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System*. Technical Report arXiv:1403.1866. arXiv, 2014. URL: http://arxiv.org/abs/1403.1866.

[FKS14c] Daniel Fett, Ralf Küsters, and Guido Schmitz. *Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web*. Technical Report arXiv:1411.7210. arXiv, 2014. URL: http://arxiv.org/abs/1411.7210.

[FKS15a] Daniel Fett, Ralf Küsters, and Guido 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 21-25, 2015, Proceedings, Part I*. Vol. 9326. Lecture Notes in Computer Science. Springer, 2015, pp. 43–65.

[FKS15b] Daniel Fett, Ralf Küsters, and Guido Schmitz. „SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM, 2015, pp. 1358–1369.

[FKS15c] Daniel Fett, Ralf Küsters, and Guido Schmitz. *SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web*. Technical Report arXiv:1508.01719. arXiv, Aug. 7, 2015. URL: http://arxiv.org/abs/1508.01719.

[FKS16a] Daniel Fett, Ralf Küsters, and Guido Schmitz. „A Comprehensive Formal Security Analysis of OAuth 2.0“. In: *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 2016, pp. 1204–1215.

[FKS16b] Daniel Fett, Ralf Küsters, and Guido Schmitz. *A Comprehensive Formal Security Analysis of OAuth 2.0*. Technical Report arXiv:1601.01229. arXiv, 2016. URL: http://arxiv.org/abs/1601.01229.

[FKS17a] Daniel Fett, Ralf Küsters, and Guido Schmitz. *The Web SSO Standard OpenID Connect: In-Depth Formal Analysis and Security Guidelines*. Technical Report arXiv:1704.08539. arXiv, 2017. URL: http://arxiv.org/abs/1704.08539.

[FKS17b] Daniel Fett, Ralf Küsters, and Guido Schmitz. „The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines“. In: *IEEE 30th Computer Security Foundations Symposium (CSF 2017)*. IEEE Computer Society, 2017.

[FKS19] Daniel Fett, Ralf Küsters, and Guido Schmitz. *SPRESSO Implementation (Demo Site and Source Code)*. 2019. URL: https://spresso.me.

[FR+07] Brad Fitzpatrick, David Recordon, et al. *OpenID Authentication 2.0*. Dec. 5, 2007. URL: http://openid.net/specs/openid-authentication-2_0.html.

[GDPR] The European Parliament and the Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679.

[Goo16] Google. *Migrating from OpenID 2.0 to OpenID Connect*. July 1, 2016. URL: https://developers.google.com/identity/protocols/OpenID2Migration.

[Gro03] Thomas Groß. „Security Analysis of the SAML Single Sign-on Browser/Artifact Profile“. In: *19th Annual Computer Security Applications Conference (ACSAC 2003)*. IEEE Computer Society, 2003, pp. 298–307.

[Han+10] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. „The Emperor's New APIs: On the (In)Secure Usage of New Client Side Primitives“. In: *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP), 2010*. 2010.

[HSN05] Steffen M. Hansen, Jakob Skriver, and Hanne Riis Nielson. „Using static analysis to validate the SAML single sign-on protocol“. In: *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005*. Ed. by Catherine Meadows. ACM, 2005, pp. 27–40.

[HBH07] Dick Hardt, Johnny Bufu, and Josh Hoyt. *OpenID Attribute Exchange 1.0 - Final*. Dec. 5, 2007. URL: https://openid.net/specs/openid-attribute-exchange-1_0.html.

[Hic15] Ian Hickson, ed. *HTML5 Web Messaging, W3C Recommendation*. May 19, 2015. URL: http://www.w3.org/TR/webmessaging/.

[HM18] Paul Hoffman and Patrick McManus. *DNS Queries over HTTPS (DoH)*. Internet-Draft ietf-doh-dns-over-https-14. Work in progress. Internet Engineering Task Force, Aug. 2018. 20 pp. URL: https://tools.ietf.org/html/draft-ietf-doh-dns-over-https-14.

[Ian16] Ian Hickson, ed. *Web Storage, W3C Recommendation*. Apr. 19, 2016. URL: http://www.w3.org/TR/2016/REC-webstorage-20160419/.

[IET09] IETF OAuth Working Group. *OAuth Security Advisory: 2009.1 – A session fixation attack against the OAuth Request Token approval flow (OAuth Core 1.0 Section 6) has been discovered*. Apr. 2009. URL: https://oauth.net/advisories/2009-1/.

[Iss13a] Mozilla Persona Issues. *Issue 3769 - Fallback IdP cookie should be a session cookie when user clicks "For this session only"*. Aug. 2013. URL: https://github.com/mozilla/persona/issues/3769.

[Iss13b] Mozilla Persona Issues. *Issue 3770 - Single-use certificates when user clicks "For this session only"*. Aug. 2013. URL: https://github.com/mozilla/persona/issues/3770.

[Iss15] W3C Subresource Integrity Issues. *Issue 21 - Consider integrity enforcement of iframe"*. Dec. 2015. URL: https://github.com/w3c/webappsec-subresource-integrity/issues/21.

[Jac02] Daniel Jackson. „Alloy: A New Technology for Software Modelling“. In: *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Vol. 2280. Lecture Notes in Computer Science. Springer, 2002, p. 20.

[Kar+07] Chris Karlof, Umesh Shankar, J. Doug Tygar, and David Wagner. „Dynamic pharming attacks and locked same-origin policies for web browsers“. In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 2007, pp. 58–71.

[Kel16] Ryan Kelly. *Shutting down persona.org in November 2016*. Mozilla Persona Notices. Jan. 12, 2016. URL: https://mail.mozilla.org/pipermail/persona-notices/2016/000005.html.

[Ker07] Florian Kerschbaum. „Simple Cross-Site Attack Prevention“. In: *Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2007*. IEEE Computer Society, 2007, pp. 464–472.

[KR00] David P. Kormann and Aviel D. Rubin. „Risks of the Passport single signon protocol“. In: *Computer Networks* 33.1-6 (2000), pp. 51–58.

[Kum12] Apurva Kumar. „Using automated model analysis for reasoning about security of web protocols“. In: *28th Annual Computer Security Applications Conference, ACSAC 2012*. Ed. by Robert H'obbes' Zakon. ACM, 2012, pp. 289–298.

[Kum14] Apurva 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*. Vol. 8688. Lecture Notes in Computer Science. Springer, 2014, pp. 192–211.

[LLW05] Fen Labalme, Mike Lindelsee, and Gabe Wachob. *An Introduction to XRIs, Working Draft 04, 14 March 2005*. Ed. by Drummond Reed and Dave McAlpin. OASIS, Mar. 14, 2005. URL: http://docs.oasis-open.org/xri/xri/V2.0/xri-intro-V2.0.pdf.

[Low95] Gavin Lowe. „An attack on the Needham-Schroeder public-key authentication protocol“. In: *Information Processing Letters* 56 (1995), pp. 131–133.

[Low96] Gavin Lowe. „Breaking and fixing the Needham-Schroeder public-key protocol using FDR“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996)*. Vol. 1055. Lecture Notes in Computer Science. Springer-Verlag, 1996, pp. 147–166.

[Mar12] Gervase Markham. *bugzilla.mozilla.org Now Supports BrowserID*. Apr. 5, 2012. URL: http://blog.gerv.net/2012/04/bugzilla-mozilla-org-browserid/.

[Mil11] "millsd". *Introducing BrowserID: A better way to sign in*. Identity at Mozilla Blog. July 14, 2011. URL: http://identity.mozilla.com/post/7616727542/introducing-browserid-a-better-way-to-sign-in. Archived version available at https://web.archive.org/web/20170521012351/http://identity.mozilla.com/post/7616727542/introducing-browserid-a-better-way-to-sign-in.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. „Z3: An Efficient SMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[MDS07] Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. „A Tutorial on Satisfiability Modulo Theories“. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 20–36.

[Moz13] Mozilla Identity Team. *BrowserID Source Code*. BrowserID Source Code Repository. Branch train-2013.03.29. Apr. 4, 2013. URL: https://github.com/mozilla/browserid.

[Moz15a] Mozilla Identity Team. *BigTent Source Code*. Oct. 20, 2015. URL: https://github.com/mozilla/persona-yahoo-bridge.

[Moz15b] Mozilla Identity Team. *Sideshow Source Code*. Mar. 16, 2015. URL: https://github.com/mozilla/persona-gmail-bridge.

[Moz18] Mozilla Identity Team. *Persona*. Mozilla Developer Network Wiki. Sept. 20, 2018. URL: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona.

[Moz19] Mozilla. *Referrer Policy*. Mozilla Developer Network Wiki. June 27, 2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[NS78] Roger M. Needham and Michael D. Schroeder. „Using Encryption for Authentication in Large Networks of Computers“. In: *Communications of the ACM* 21.12 (1978), pp. 993–999.

[Ope17] Open Web Application Security Project (OWASP). *Session Management Cheat Sheet*. Nov. 9, 2017. URL: https://www.owasp.org/index.php/Session_Management_Cheat_Sheet.

[Ope18] Open Web Application Security Project (OWASP). *Password Storage Cheat Sheet*. Mar. 18, 2018. URL: https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet.

[Pai+11] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. „Formal Verification of OAuth 2.0 Using Alloy Framework“. In: *CSNT '11 Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*. Proceedings of the International Conference on Communication Systems and Network Technologies, 2011, pp. 655–659.

[RFC1035] P.V. Mockapetris. *Domain names - implementation and specification*. RFC 1035 (Internet Standard). RFC. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766. Fremont, CA, USA: RFC Editor, Nov. 1987. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/rfc/rfc1035.txt.

[RFC4033] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. *DNS Security Introduction and Requirements*. RFC 4033 (Proposed Standard). RFC. Updated by RFCs 6014, 6840. Fremont, CA, USA: RFC Editor, Mar. 2005. DOI: 10.17487/RFC4033. URL: https://www.rfc-editor.org/rfc/rfc4033.txt.

[RFC4120] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. *The Kerberos Network Authentication Service (V5)*. RFC 4120 (Proposed Standard). RFC. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751, 8062, 8129. Fremont, CA, USA: RFC Editor, July 2005. DOI: 10.17487/RFC4120. URL: https://www.rfc-editor.org/rfc/rfc4120.txt.

[RFC5785] M. Nottingham and E. Hammer-Lahav. *Defining Well-Known Uniform Resource Identifiers (URIs)*. RFC 5785 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Apr. 2010. DOI: 10.17487/RFC5785. URL: https://www.rfc-editor.org/rfc/rfc5785.txt.

[RFC5849] E. Hammer-Lahav (Ed.) *The OAuth 1.0 Protocol*. RFC 5849 (Informational). RFC. Obsoleted by RFC 6749. Fremont, CA, USA: RFC Editor, Apr. 2010. DOI: 10.17487/RFC5849. URL: https://www.rfc-editor.org/rfc/rfc5849.txt.

[RFC6265] A. Barth. *HTTP State Management Mechanism*. RFC 6265 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Apr. 2011. DOI: 10.17487/RFC6265. URL: https://www.rfc-editor.org/rfc/rfc6265.txt.

[RFC6394] R. Barnes. *Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)*. RFC 6394 (Informational). RFC. Fremont, CA, USA: RFC Editor, Oct. 2011. DOI: 10.17487/RFC6394. URL: https://www.rfc-editor.org/rfc/rfc6394.txt.

[RFC6749] D. Hardt (Ed.) *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). RFC. Updated by RFC 8252. Fremont, CA, USA: RFC Editor, Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/rfc/rfc6749.txt.

[RFC6797] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*. RFC 6797 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Nov. 2012. DOI: 10.17487/RFC6797. URL: https://www.rfc-editor.org/rfc/rfc6797.txt.

[RFC7230] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7230. URL: https://www.rfc-editor.org/rfc/rfc7230.txt.

[RFC7231] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7231. URL: https://www.rfc-editor.org/rfc/rfc7231.txt.

[RFC7232] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7232. URL: https://www.rfc-editor.org/rfc/rfc7232.txt.

[RFC7233] R. Fielding (Ed.), Y. Lafon (Ed.), and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. RFC 7233 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7233. URL: https://www.rfc-editor.org/rfc/rfc7233.txt.

[RFC7234] R. Fielding (Ed.), M. Nottingham (Ed.), and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7234. URL: https://www.rfc-editor.org/rfc/rfc7234.txt.

[RFC7235] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC 7235 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7235. URL: https://www.rfc-editor.org/rfc/rfc7235.txt.

[RFC791] J. Postel. *Internet Protocol*. RFC 791 (Internet Standard). RFC. Updated by RFCs 1349, 2474, 6864. Fremont, CA, USA: RFC Editor, Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/rfc/rfc791.txt.

[RFC793] J. Postel. *Transmission Control Protocol*. RFC 793 (Internet Standard). RFC. Updated by RFCs 1122, 3168, 6093, 6528. Fremont, CA, USA: RFC Editor, Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/rfc/rfc793.txt.

[RFC8446] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt.

[SS84] S. Goldwasser and S. Micali. „Probabilistic Encryption“. In: *Journal of Computer and System Sciences* 28.2 (Apr. 1984), pp. 270–299.

[SSR88] S. Goldwasser, S. Micali, and R.L. Rivest. „A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks“. In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308.

[Sak+14] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. Nov. 8, 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.

[Rag+08] Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo, eds. *SAML 2.0 Technical Overview. Committee Draft 02*. Mar. 25, 2008. URL: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.html.

[Som+12] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. „On Breaking SAML: Be Whoever You Want to Be“. In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. Ed. by Tadayoshi Kohno. USENIX Association, 2012, pp. 397–412.

[SS13] Sooel Son and Vitaly Shmatikov. „The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites“. In: *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[SKS10] Pavol Sovis, Florian Kohlar, and Jörg Schwenk. „Security Analysis of OpenID“. In: *Sicherheit*. Vol. 170. LNI. GI, 2010, pp. 329–340.

[Sto+17] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. „How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security“. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 971–987.

[SB12] San-Tsai Sun and Konstantin Beznosov. „The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems“. In: *ACM Conference on Computer and Communications Security, CCS'12*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 378–390.

[SHB12] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. „Systematically Breaking and Fixing OpenID Security: Formal Analysis, Semi-Automated Empirical Evaluation, and Practical Countermeasures“. In: *Computers & Security* 31.4 (2012), pp. 465–483.

[Swa+13] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. „Secure distributed programming with value-dependent types“. In: *J. Funct. Program.* 23.4 (2013), pp. 402–451.

[Ter18] Brian Terlson, ed. *ECMAScript® 2018 Language Specification*. ECMA-262. Approved as international standard ISO/IEC 16262. 2018. URL: http://www.ecma-international.org/ecma-262/9.0/index.html.

[TB15] Shane Tomlinson and Marcel Bokhorst. *Mozilla Persona (BrowserID) plugin for Wordpress*. 2015. URL: https://wordpress.org/plugins/browserid/.

[WCW12] Rui Wang, Shuo Chen, and XiaoFeng 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*. IEEE Computer Society, 2012, pp. 365–379.

[Wan+11] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. „How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores“. In: *32nd IEEE Symposium on Security and Privacy, S&P 2011*. IEEE Computer Society, 2011, pp. 465–480.

[WHA19a] WHATWG. *Fetch*. Living Standard. Mar. 23, 2019. URL: https://fetch.spec.whatwg.org/.

[WHA19b] WHATWG. *HTML*. Living Standard. Apr. 15, 2019. URL: https://html.spec.whatwg.org/.

[WHA19c] WHATWG. *XMLHttpRequest*. Living Standard. Mar. 24, 2019. URL: https://xhr.spec.whatwg.org/.

[YHR04] Tom Yu, Sam Hartman, and Kenneth Raeburn. „The Perils of Unauthenticated Encryption: Kerberos Version 4“. In: *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2004.

[Zhe+15] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. „Cookies Lack Integrity: Real-World Implications“. In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, Aug. 2015, pp. 707–721. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng.

[ZE14] Yuchen Zhou and David 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*. USENIX Association, 2014, pp. 495–510.

Academic Curriculum and Publications

Academic Curriculum

Jan. 2017 –	**University of Stuttgart, Germany.**
	Ph.D. student at the Institute of Information Security.
	Supervisor: Prof. Dr. Ralf Küsters
Aug. 2012 – Jan. 2017	**University of Trier, Germany.**
	Ph.D. student at the Chair of Information Security and Cryptography.
	Supervisor: Prof. Dr. Ralf Küsters
Apr. 2005 – Aug. 2012	**University of Trier, Germany.**
	Diploma in Computer Science.
	Thesis title: *Analysis of a Digital Locking System*
	Supervisor: Prof. Dr. Ralf Küsters

Publications

- Daniel Fett, Ralf Küsters, and Guido Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *IEEE 30th Computer Security Foundations Symposium (CSF 2017)*. IEEE Computer Society, 2017.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. *The Web SSO Standard OpenID Connect: In-Depth Formal Analysis and Security Guidelines*. Technical report arXiv:1704.08539. arXiv, 2017. URL: http://arxiv.org/abs/1704.08539.

- Daniel Fett and Guido Schmitz. "Pi and More - eine Veranstaltungsreihe rund um ‚kleine Computer'". In: *46. Jahrestagung der Gesellschaft für Informatik, Informatik 2016, 26.–30. September 2016, Klagenfurt, Österreich*. Vol. P-259. LNI. GI, 2016, pp. 1195–1196.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 2016, pp. 1204–1215.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. *A Comprehensive Formal Security Analysis of OAuth 2.0*. Technical report arXiv:1601.01229. arXiv, 2016. URL: http://arxiv.org/abs/1601.01229.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. "SPRESSO: A Secure, Privacy- Respecting Single Sign-On System for the Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM, 2015, pp. 1358–1369.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. *SPRESSO: A Secure, Privacy- Respecting Single Sign-On System for the Web*. Technical report arXiv:1508.01719. arXiv, Aug. 7, 2015. URL: http://arxiv.org/abs/1508.01719.

- Daniel Fett, Ralf Küsters, and Guido 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 21-25, 2015, Proceedings, Part I*. Vol. 9326. Lecture Notes in Computer Science. Springer, 2015, pp. 43–65.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. *Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web*. Technical report arXiv:1411.7210. arXiv, 2014. URL: http://arxiv.org/abs/1411.7210.

- Daniel Fett, Ralf Küsters, and Guido 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)*. IEEE Computer Society, 2014, pp. 673–688.

- Daniel Fett, Ralf Küsters, and Guido Schmitz. *An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System*. Technical report arXiv:1403.1866. arXiv, 2014. URL: http://arxiv.org/abs/1403.1866.