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

Daniel Fett
University of Trier, Germany
fett@uni-trier.de

Ralf Küsters
University of Trier, Germany
kuesters@uni-trier.de

Guido Schmitz
University of Trier, Germany
schmitzg@uni-trier.de

ABSTRACT

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

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

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

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

1. INTRODUCTION

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

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

The first system so far which was designed with the intent to respect users' privacy was the BrowserID system [18, 19], which is a relatively new system developed by Mozilla and is also known by its marketing name *Persona*.

Unfortunately, in [11] severe attacks against BrowserID were discovered, which show that the privacy of BrowserID is completely broken: these attacks allow malicious IdPs and in some versions of the attacks even arbitrary parties to check the login status of users at any RP with little effort (see Section 2.1 for some more details on these attacks). Even worse, these attacks exploit design flaws of BrowserID that, as discussed in [11], cannot be fixed without a major redesign of the system, and essentially require building a new system. As further discussed in Section 2.4, besides the lack of privacy there are also other issues that motivate the design of a new system.

The goal of this work is therefore to design the (first) SSO system which respects users' privacy in the sense described above, i.e., IdPs (even completely malicious ones) should not be able to track at which RPs users log in. Moreover, the history of SSO systems shows that it is highly non-trivial to design secure SSO systems, not only w.r.t. privacy requirements, but even w.r.t. authentication requirements. Attacks easily go unnoticed and in fact numerous attacks on SSO systems, including attacks on OAuth, OpenID, Google ID, Facebook Connect, SAML, and BrowserID have been uncovered which compromise the security of many services and users at once [4–6, 20, 21, 24–27]. Besides designing and implementing a privacy-respecting SSO system, we therefore also carry out a formal security analysis of the system based on an expressive model of the web infrastructure in order to provide formal security guarantees. More specifically, the contributions of our work are as follows.

Contributions of this Paper. In this work, we propose the system SPRESSO (for Secure Privacy-REspecting Single Sign-On). This

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCS'15, October 12–16, 2015, Denver, Colorado, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813726.

is the first SSO system which respects user's privacy. The system allows users to log in to RPs with their email addresses. A user is authenticated to an RP by the IdP hosting the user's email address. This is done in such a way that the IdP does not learn at which RP the user wants to log in.

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

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

We formally prove that SPRESSO enjoys strong authentication and privacy properties. Our analysis is based on an expressive Dolev-Yao style model of the web infrastructure [10]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as new technologies, such as web storage and cross-document messaging (postMessages). JavaScript is modeled in an abstract way by so-called scripting processes which can be sent around and, among others, can create iframes and initiate XMLHTTPRequests (XHRs). Browsers may be corrupted dynamically by the adversary.

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

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

Structure of this Paper. In Section 2, we describe our system and discuss and motivate design choices. We then, in Section 3, briefly recall the general web model from [10] and explain the modifications and extensions we made. The mentioned strong but general definition of indistinguishability/privacy for web applications is presented in Section 4. In Section 5, we provide the formal model of SPRESSO, based on which we state and analyze privacy and authentication of SPRESSO in Sections 6 and 7, respectively. Further related work is discussed in Section 8. We conclude in Section 9. All details and proofs are available in our technical report [12]. An online demo and the source code of SPRESSO are available at [22].

2. DESCRIPTION OF SPRESSO

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

2.1 Main Features

SPRESSO enjoys the following key features:

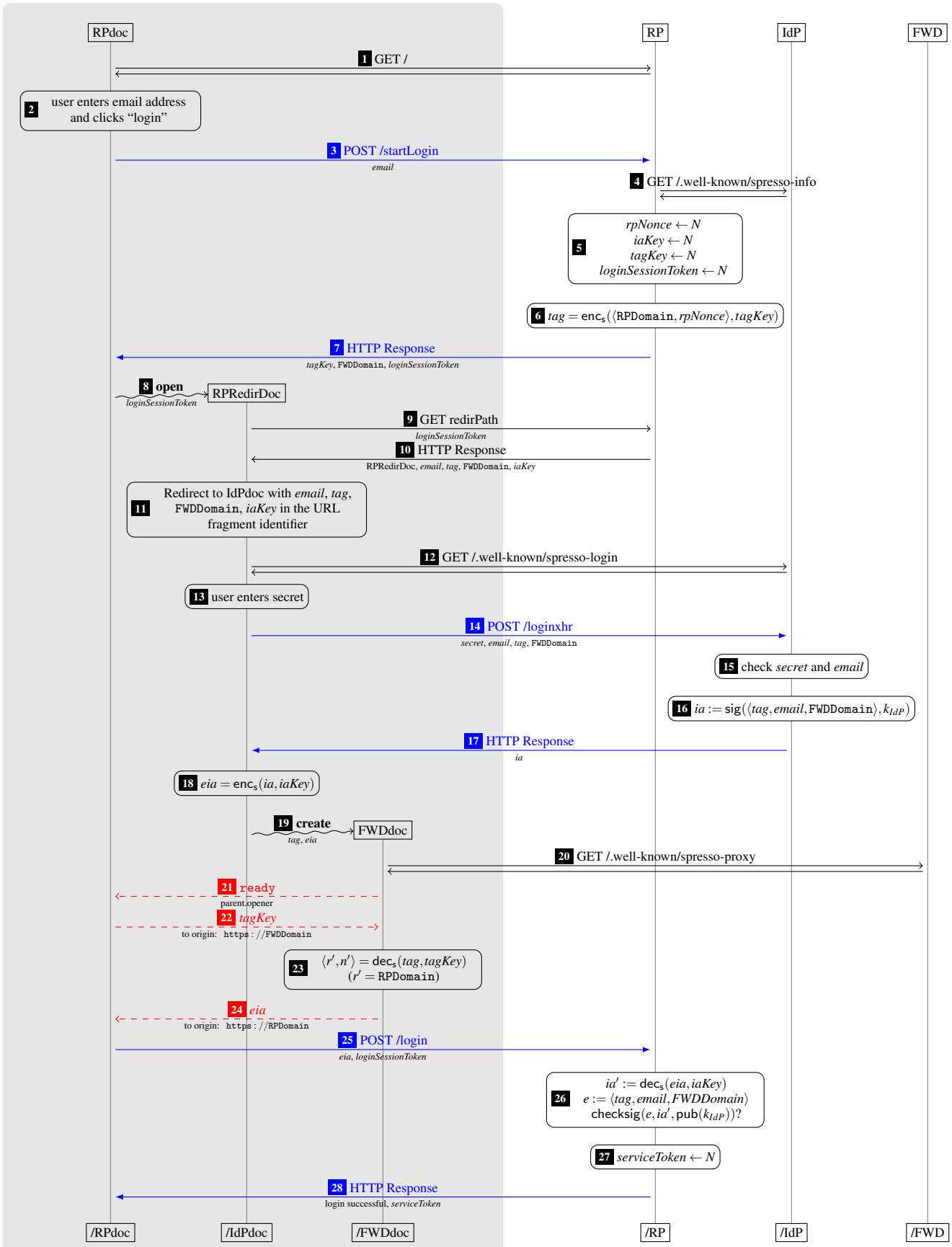
Strong Authentication and Privacy. SPRESSO is designed to satisfy strong authentication and privacy properties.

Authentication is the most fundamental security property of an SSO system. That is, i) an adversary should not be able to log in to an RP, and hence, use the service of the RP, as an honest user, and ii) an adversary should not be able to log in the browser of an honest user under an adversary's identity (identity injection). Depending on the service provided by the RP, a violation of ii) could allow the adversary to track an honest user or to obtain user secrets. We note that in the past, attacks on authentication have been found in almost all deployed SSO systems (e.g., OAuth, OpenID, and BrowserID [10, 11, 21, 23, 24, 26, 27]).

While authentication assumes the involved RP and IdP to be honest, privacy is concerned with malicious IdPs. This property requires that (malicious) IdPs should not be able to track at which RPs specific users log in. As already mentioned in the introduction, so far, except for BrowserID, no other SSO system was designed to provide privacy. (In fact, exchanging user data between IdPs and RPs directly is a key part of the protocols in OpenID and OAuth, for example, and hence, in such protocols, IdPs can easily track at which RP a user logs in.) However, BrowserID failed to provide privacy: As shown in [11], a subtle attack allowed IdPs (and in some versions of the attack even arbitrary parties) to check the login status of users at any RP. More specifically, by running a malicious JavaScript within the user's browser, an IdP can, for any RP, check whether the user is logged in at that RP by triggering the (automatic) login process and testing whether a certain iframe is created during this process or not. The (non-)existence of this iframe immediately reveals the user's login status. Hence, a malicious IdP can track at which RP a user is logged in. As we discuss in [11], this could not be fixed without a major redesign of BrowserID. Our work could be considered such a major redesign. While SPRESSO shares some basic concepts with BrowserID, SPRESSO is, however, not based on BrowserID, but a new system built from scratch (see the discussion in Section 2.4).

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

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



⟶ HTTPS messages, → XHRs (over HTTPS), - -→ postMessages, ⇝ browser commands

Figure 1: SPRESSO Login Flow.

port, a seamless fallback can be provided, as discussed later.

Adherence to Web Standards. SPRESSO is based solely on standard HTML5 and web features and uses no browser extensions, plug-ins, or other client-side executables. This guarantees that SPRESSO can be used across browsers, platforms and devices, including both desktop computers and mobile platforms, without installing any software (besides a browser). Note that on smartphones, for example, browsers usually do no support extensions or plug-ins.

2.2 Login Flow

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

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

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

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

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

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

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

After the browser loaded IdPdoc from IdP, the user enters her password[2] matching her email address [13]. The password, the email address, the tag, and the FWD domain are now sent to IdP [14]. After IdP verified the user credentials [15], it creates the identity assertion as the signature

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

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

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

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

Then, IdPdoc opens an iframe with the URL of FWDdoc, passing the tag and the encrypted IA to FWDdoc. After the iframe is loaded [20], FWDdoc sends a postMessage[3] to its parent's opener window, which is RPdoc [21]. This postMessage with the sole content "ready" triggers RPdoc to send the *tagKey* to FWDdoc, where in the postMessage the origin[4] of FWD with HTTPS is declared to be the only allowed receiver of this message [22]. FWDdoc uses the key to decrypt the tag and thereby learns the intended receiver (RP) of the IA [23]. As its last action, FWD forwards the encrypted IA *eia* via postMessage to RPdoc (using RP's HTTPS origin as the only allowed receiver) [24].

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

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

2.3 Implementation Details

We developed a proof-of-concept implementation of SPRESSO in about 700 lines of JavaScript and HTML code. It contains all

[1] 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.

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

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

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

presented features of SPRESSO itself and a typical IdP. The implementation (source code and online demo) is available at [22]. Our model presented in Section 5 closely follows this implementation.

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

2.4 Discussion

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

Malicious RP: Impersonation Attack. An attacker could try to launch a man in the middle attack against SPRESSO by playing the role of an RP (RP server and RPdoc) to the user. Such an attacker would run a malicious server at his RP domain, say, `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 $\mathsf{enc_s}(\langle \texttt{RPb}, rpNonce\rangle, tagKey)$ 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 24. Our formal analysis presented in the following sections indeed proves that such attacks are excluded in SPRESSO. We note that in order to make sure that the postMessage is delivered to the correct RP window (technically, a window with the expected origin), FWD uses a standard feature of the postMessage mechanism which allows to specify the origin of the intended recipient of a postMessage.

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

The IdP could try to undermine the user's privacy by trying to find out which RP requests the IA. However, in SPRESSO, the IdP cannot gather such information: From the information available to it (*email*, *tag*, *FWDDomain* plus any information it can gather from the browser's state), it cannot infer the RP.[5] It could further try to 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 Sections 4 and 6).

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

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

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

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

[5] 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.

[6] 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.

specific user logs in. (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.)

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

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

A native web technology called *subresource integrity* (SRI)[7] is currently under development at the W3C. SRI allows a document to create an iframe with an attribute *integrity* that takes a hash value. The browser now would guarantee that the document loaded into the iframe hashes to exactly the given value. So, essentially the creator of the iframe can enforce the iframe to be loaded with a specific document. This would enable SPRESSO to automatically check the integrity of FWDdoc without any extensions.

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

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

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

Tag Length Side Channel. The length of the tag created in Step 6 depends on the length of `RPDomain`. Since the tag is given to IdP, IdP might try to infer `RPDomain` from the length of the tag. However, according to RFC 1035, domain names may at most be 253 characters long. Therefore, by appropriate padding (e.g., encrypting always nine 256 Bit plaintext blocks)[9] the length of the tag will not reveal any information about `RPDomain`.

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

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

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

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

[7] `http://www.w3.org/TR/SRI/`

[8] 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.

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

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

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.

3. WEB MODEL

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

Here, we only present a very brief version of the web model. The full model, including our changes, is provided in our technical report [12].

3.1 Communication Model

The main entities in the communication model are *atomic processes*, which are used to model web browsers, web servers, DNS servers as well as web and network attackers. Each atomic process listens to one or more (IP) addresses. A set of atomic processes forms what is called a *system*. Atomic processes can communicate via events, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from the current "pool" of events and is delivered to one of the atomic processes that listens to the receiver address of that event. The atomic process can then process the event and output new events, which are added to the pool of events, and so on. More specifically, messages, processes, etc. are defined as follows.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Web System

A web system formalizes the web infrastructure and web applications. Formally, a *web system* is a tuple $(\mathcal{W},\mathcal{S},\mathsf{script},E^0)$ with the following components:

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

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

3.3 Web Browsers

We now sketch the model of the web browser, with full details provided in [12]. A web browser is modeled as a DY process (I^p,Z^p,R^p,s_0^p).

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

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

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

Figure 2 shows a brief overview of the browser relation R^p which defines how browsers behave. For example, when a `TRIGGER` message is delivered to the browser, the browser non-deterministically choses an *action*. If, for instance, this action is 1, then an active document is selected non-deterministically, and its script is triggered. The script (with inputs as outlined above), can now output a command, for example, to follow a hyperlink (`HREF`). In this case, the browser will follow this link by first creating a new DNS request. Once a response to that DNS request arrives, the actual HTTP request (for the URL defined by the script) will be sent out. After a response to that HTTP request arrives, the browser creates a new document from the contents of the response. Complex navigation and security rules ensure that scripts can only manipulate specific aspects of the browser's state. Browsers can become corrupted, i.e., be taken over by web and network attackers. The browser model comprises two types of corruption: *close-corruption*, modeling that a browser is closed by the user, and hence, certain data is removed (e.g., session cookies and opened windows), before it is taken over by the attacker, and *full corruption*, where no data is removed in advance. Once corrupted, the browser behaves like an attacker process.

4. INDISTINGUISHABILITY OF WEB SYSTEMS

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

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

PROCESSING INPUT MESSAGE m

$m =$ FULLCORRUPT: *isCorrupted* := FULLCORRUPT

$m =$ CLOSECORRUPT: *isCorrupted* := CLOSECORRUPT

$m =$ TRIGGER: non-deterministically choose *action* from $\{1,2,3\}$

action $= 1$: Call script of some active document. Outputs new state and *command*.

command = HREF: $\rightarrow$ *Initiate request*

command = IFRAME: Create subwindow, $\rightarrow$ *Initiate request*

command = FORM: $\rightarrow$ *Initiate request*

command = SETSCRIPT: Change script in given document.

command = SETSCRIPTSTATE: Change state of script in given document.

command = XMLHTTPREQUEST: $\rightarrow$ *Initiate request*

command = BACK or FORWARD: Navigate given window.

command = CLOSE: Close given window.

command = POSTMESSAGE: Send postMessage to specified document.

action $= 2$: $\rightarrow$ *Initiate request to some URL in new window*

action $= 3$: $\rightarrow$ *Reload some document*

$m =$ DNS response: send corresponding HTTP request

$m =$ HTTP(S) response: (decrypt,) find reference.

reference to window: create document in window

reference to document: add response body to document's script input

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

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

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

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

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

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 command, if any, is processed (deterministically) by the browser. If the script to be executed is an honest script (i.e., not R^{att}), then this script is evaluated and the resulting command is processed by the browser. (Note that the script might perform non-deterministic actions.) iv) If p is an honest process (but not a browser), then the process evaluates e as usual. (Again, the computation might be non-deterministic, as honest processes might be non-deterministic.)

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

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

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

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

We also need the following terminology. If $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a web system and p is an attacker process in $\mathcal{W}$, then 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. In our indistinguishability definition, we will consider the state of the distinguished attacker process only. This is sufficient since the attacker can send all its data to this process.

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

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

5. FORMAL MODEL OF SPRESSO

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

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

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

We now sketch the processes and the scripts in $\mathcal{W}$ and $\mathcal{S}$ (see [12] for full details). As mentioned, our modeling closely follows the description in Section 2 and the implementation of the system:

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

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

6. PRIVACY OF SPRESSO

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

Definition of Privacy of SPRESSO. The web systems considered for the privacy of SPRESSO are the web systems $\mathcal{SWS}$ defined in Section 5 which now contain one or more web attackers, no network attackers, one honest DNS server, one honest forwarder, one browser, and two honest relying parties r_1 and r_2. All honest parties may not become corrupted and use the honest DNS server for address resolving. Identity providers are assumed to be dishonest, and hence, are subsumed by the web attackers (which govern all identities). The web attacker subsumes also potentially dishonest forwarders, DNS servers, relying parties, and other servers. The honest relying parties are set up such that they already contain the public signing keys (used to verify identity assertions) for each domain registered at the DNS server, modeling that these have been cached by the relying parties, as discussed in Section 2.2.

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

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

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

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

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

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

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

THEOREM 1. *SPRESSO is IdP-private.*

The full proof is provided in our technical report [12]. In the proof, we define an equivalence relation between configurations of $\mathcal{SWS}^{\mathit{priv}}(r_1)$ and $\mathcal{SWS}^{\mathit{priv}}(r_2)$, comprising equivalences between states and equivalences between events (in the pool of waiting

events). For the states, for each (type of an) atomic DY process in the web system, we define how their states are related. For example, the state of the FWD server must be identical in both configurations. As another example, roughly speaking, the attacker's state is the same up to subterms the attacker cannot decrypt. Regarding (waiting) events, we distinguish between messages that result (directly or indirectly) from a `CHALLENGE` request by the browser and other messages. While the challenged messages may differ in certain ways, other messages may only differ in parts that the attacker cannot decrypt.

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

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

7. AUTHENTICATION OF SPRESSO

We show that SPRESSO satisfies two fundamental authentication properties.

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

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

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

We call a web system $\mathcal{SWS}^{auth}$ *secure w.r.t. authentication* if the above conditions are satisfied in all runs of the system. We refer the reader to our technical report [12] for the formal definition of (A) and (B).

Analyzing Authentication of SPRESSO. We prove the following theorem:

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

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

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

We then use these properties and the general web system properties shown in the full version of [11] to prove properties (A) and (B) separately. In both cases, we assume that the respective property is not satisfied and lead this to a contradiction. Again, the full proof is provided in our technical report [12].

8. FURTHER RELATED WORK

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

Besides the design and implementation of SPRESSO, the formal analysis of this system based on an expressive web model is an important part of our work. The formal treatment of the security of web applications is a young discipline. Of the few works in this area even less are based on a general model that incorporates essential mechanisms of the web. Early works in formal web security analysis (see, e.g., [3, 8, 15, 16, 24]) are based on very limited models developed specifically for the application under scrutiny. The first work to consider a general model of the web, written in the finite-state model checker Alloy, is the work by Akhawe et al. [2]. Inspired by this work, Bansal et al. [5, 6] built a more expressive model, called WebSpi, in ProVerif [7], a tool for symbolic cryptographic protocol analysis. These models have successfully been applied to web standards and applications. Recently, Kumar [17] presented a high-level Alloy model and applied it to SAML single sign-on. The web model presented in [10], which we further extend and refine here, is the most comprehensive web model to date (see also the discussion in [10]). In fact, this is the only model in which we can analyze SPRESSO. For example, other models do not incorporate a precise handling of windows, documents, or iframes; cross-document messaging (postMessages) are not included at all.

9. CONCLUSION

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

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

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

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

10. REFERENCES

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL 2001*, pages 104–115. ACM Press, 2001.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF 2010*, pages 290–304. IEEE Computer Society, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In *FMSE 2008*, pages 1–10. ACM, 2008.

[4] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS'13*. The Internet Society, 2013.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *POST 2013*, volume 7796 of *LNCS*, pages 126–146. Springer, 2013.

[6] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF 2012*, pages 247–262. IEEE Computer Society, 2012.

[7] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW-14*, pages 82–96. IEEE Computer Society, 2001.

[8] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[9] V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: negative tests and non-determinism. In *CCS 2011*, pages 321–330. ACM, 2011.

[10] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *S&P 2014*, pages 673–688. IEEE Computer Society, 2014.

[11] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *ESORICS 2015*, *LNCS*. Springer, 2015. To appear. Full version available at http://arxiv.org/abs/1411.7210.

[12] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. Technical Report arXiv:1508.01719, arXiv, 2015. Available at http://arxiv.org/abs/1508.01719.

[13] B. Fitzpatrick, D. Recordon, et al. OpenID Authentication 2.0. Dec. 5, 2007. http://openid.net/specs/openid-authentication-2_0.html.

[14] D. Hardt. RFC6749 - The OAuth 2.0 Authorization Framework. Oct. 2012. http://tools.ietf.org/html/rfc6749.

[15] D. Jackson. Alloy: A New Technology for Software Modelling. In *TACAS 2002*, volume 2280 of *LNCS*, page 20. Springer, 2002.

[16] F. Kerschbaum. Simple Cross-Site Attack Prevention. In *SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.

[17] A. Kumar. A Lightweight Formal Approach for Analyzing Security of Web Protocols. In *RAID 2014*, volume 8688 of *LNCS*, pages 192–211. Springer, 2014.

[18] Mozilla Identity Team. Persona. https://login.persona.org.

[19] T. Nitot. Persona: more privacy, better security while making developers and users happy! Beyond the Code Blog. Apr. 9, 2013. https://blog.mozilla.org/beyond-the-code/2013/04/09/persona-beta2/.

[20] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *USENIX 2012*, pages 397–412. USENIX Association, 2012.

[21] P. Sovis, F. Kohlar, and J. Schwenk. Security Analysis of OpenID. In *Sicherheit*, volume 170 of *LNI*, pages 329–340. GI, 2010.

[22] SPRESSO Demo Site and Source Code, 2015. https://spresso.me.

[23] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *CCS'12*, pages 378–390. ACM, 2012.

[24] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically Breaking and Fixing OpenID Security: Formal Analysis, Semi-Automated Empirical Evaluation, and Practical Countermeasures. *Computers & Security*, 31(4):465–483, 2012.

[25] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *S&P 2012*, pages 365–379. IEEE Computer Society, 2012.

[26] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX 2013*, pages 399–314. USENIX Association, 2013.

[27] Y. Zhou and D. Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *USENIX 2014*, pages 495–510. USENIX Association, 2014.