UPPRESSO: Untraceable and Unlinkable Privacy-PREserving Single Sign-On Services

Abstract

Single sign-on (SSO) allows a user to maintain only the credential at the identity provider (IdP), instead of one credential for each relying party (RP), to login to numerous RPs. However, SSO introduces extra privacy leakage threats, compared with traditional authentication mechanisms, as (*a*) the IdP could track all the RPs which a user is visiting, and (*b*) collusive RPs could learn a user's online profile by linking his identities across these RPs. Several privacy-preserving SSO solutions have been proposed to defend against either the curious IdP or collusive RPs, but none of them addresses both of these privacy leakage threats at the same time.

In this paper, we propose a privacy-preserving SSO system, called *UPPRESSO*, to protect a user's login traces against both the curious IdP and collusive RPs simultaneously. We analyze the identity dilemma between the SSO security requirements and these privacy concerns, and convert the SSO privacy problems into an identity-transformation challenge. In each login instance of UPPRESSO, an *ephemeral pseudo-identity* (denoted as PID_{RP}) of the RP which the user is attempting to visit, is firstly negotiated between the RP and the user. Then, PID_{RP} is sent to the IdP and designated in the identity token, so that the IdP is not aware of the visited RP. Meanwhile, PID_{RP} is used by the IdP to transform the *permanent user identity* ID_U into an *ephemeral user pseudo-identity* (denoted as PID_U) in the identity token. On receiving the identity token, the RP transforms PID_U into a *permanent account* (denoted as *Acct*) of the user, by a trapdoor in the negotiation. Given a user, the account at each RP is unique and different from ID_U, so collusive RPs cannot link his identities across multiple RPs. To the best of our knowledge, this is the first practical SSO solution which solves the privacy problems caused by both the curious IdP and collusive RPs.

We build the UPPRESSO prototype system for web applications, with standard functions of OpenID Connect (OIDC): the function of RP Dynamic Registration is used to support ephemeral PID_{RP}, while the function of Core Sign-On is slightly improved to calculate PID_U and *Acct*. The prototype system is implemented on top of open-source MITREid Connect, and the extensive evaluation shows that UPPRESSO introduces reasonable overheads and fulfills the requirements of both security and privacy.

1 Introduction

Single sign-on (SSO) systems such as OpenID Connect (OIDC) [1], OAuth 2.0 [2] and SAML [3], are widely deployed in the Internet for identity management and authentication. With the help of SSO, a user logins to a website, referred to as the *relying party* (RP), using his identity registered at another web service, known as the *identity provider* (IdP). An RP delegates user identification and authentication to the trusted IdP, which issues an *identity token* (e.g., id token in OIDC or identity assertion in SAML) for a user to visit the RP after authenticating this user. Thus, the user keeps only one credential for the IdP, instead of maintaining several credentials for different RPs.

The SSO login flow for web applications works as below. For example, in the widely-used OIDC systems, a user sends a login request to the target RP, and the RP constructs an identity-token request with its identity (denoted as ID_{RP} in this paper) and redirects the request to the IdP. After authenticating the user, the IdP issues an identity token binding the identities of both the user and the RP (i.e., ID_U and ID_{RP}), which is returned to the user and forwarded to the RP. Finally, the RP verifies the identity token to decide whether the user is allowed to login or not.

The wide adoption of SSO raises concerns on user privacy [4–7], because SSO facilitates adversaries to track which RPs a user visits. In order to issue identity tokens, in each login instance the IdP is always aware of when and to which RP a user attempts to login. As a result, a curious-but-honest IdP could track all the RPs that each user has visited over time [6, 7], called the *IdP-based login tracing* in this paper. Meanwhile, the RPs learn users identities from the identity tokens. If the IdP encloses a unique user identity in the tokens for a user to visit different RPs [8, 9], collusive RPs could link these login instances across the RPs, to learn his online

profile [4]. We denote this privacy risk as the *RP-based identity linkage*. All existing SSO protocols leak user privacy in different ways, and it is worth noting that *these two kinds of privacy threats result from the designs of SSO protocols* [5], but not any specific implementations of SSO systems.

Several solutions have been proposed to protect user privacy in SSO services [4–7]. However, to the best of our knowledge, *none of them provides a practical protection against both the IdP-based login tracing and the RP-based identity linkage* (see Sections 2.2 and 2.3 for details). The techniques proposed so far to defend against each of the two threats cannot be integrated, for they require modifications to the SSO login flows that essentially conflicts. This requires a non-trivial re-design of SSO protocols against the threats of user privacy, while providing secure identity management and authentication.

In this paper, we conceptualize the privacy requirements of SSO into an *identity transformation problem*, and propose an Untraceable and Unlinkable Privacy-PREserving Single Sign-On (UPPRESSO) protocol to comprehensively protect user privacy. In particular, we design three identity-transformation functions in the SSO login flow. In each login instance of UPPRESSO, ID_{RP} is firstly transformed to an ephemeral PID_{RP} cooperatively by the RP and the user. Then, PID_{RP} is sent to the IdP to transform ID_U to ephemeral PID_U, so that the identity token binds PID_U and PID_{RP}, instead of permanent ID_U and ID_{RP}. Finally, after receiving an identity token with matching PID_{RP}, the RP transforms PID_U into an account. Given a user, this account (*a*) is unique at each RP but (*b*) keeps permanent across multiple login instances. While providing non-anonymous SSO services, UPPRESSO prevents the IdP from tracking a user's login activities because it receives only PID_{RP} in the identity-token request, and collusive RPs from linking a user's accounts across these RPs because every account is unique.

We summarize our contributions as follows.

- We formalize the SSO privacy problems as an identity-transformation challenge and analyze the limitations of existing privacy-preserving SSO designs.
- We propose a comprehensive solution to protect the users' login activities against the curious IdP and collusive RPs; that is, solve this challenge effectively by designing identity-transformation functions. To the best of our knowledge, UPPRESSO is the first practical SSO system against both the IdP-based login tracing and the RP-based identity linkage.
- We implement the UPPRESSO prototype system for web applications, based on open-source MITREid Connect. The performance evaluations show that UPPRESSO introduces reasonable overheads.

The remainder is organized as follows. Section 2 presents the background and related works. The identity dilemma of privacy-preserving SSO is analyzed in Section 3, and Section 4 presents the detailed design of UPPRESSO. The properties of security and privacy are analyzed in Section 5. We explain the prototype implementation and experimental evaluations in Section 6, and discuss some extended issues in Section 7. Section 8 concludes this work.

2 Background and Related Works

This section introduces OIDC [1], to explain typical SSO login flows. Then, we discuss existing privacy-preserving SSO solutions and other related works.

2.1 OpenID Connect

OIDC is one of the most popular SSO protocols for web applications. Users and RPs initially register at the IdP with their identities and other necessary information such as user credentials (i.e., passwords or public keys) and RP endpoints (i.e., the URLs to receive identity tokens).

Implicit Login Flow. OIDC supports three types of user login flows: implicit flow, authorization code flow, and hybrid flow (i.e., a mix-up of the previous two). These flows mainly result from the different steps to request and receive identity tokens, but work with the common security requirements of identity tokens. We introduce the implicit flow and present our design and implementation based on this flow, and the extensions to support the authorization code flow is discussed in Section 7.

As shown in Figure 1, a user firstly initiates a login request to an RP. Then, the RP constructs an identity-token request with its own identity and the scope of requested user attributes. This identity-token request is redirected to the IdP. After authenticating the user, the IdP issues an identity token which is forwarded by the user to the RP endpoint. The token contains a user identity (or pseudo-identity), the RP identity, a validity period, the requested user attributes, etc. Finally, the RP verifies the received identity token and allows the user to login as the identity enclosed in this token.

Before issuing the identity token, the IdP obtains the user's authorization to enclose the requested user attributes. The

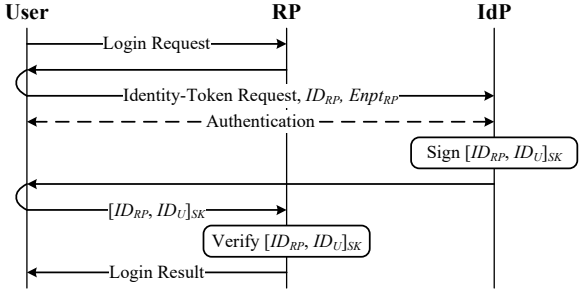

Figure 1: The implicit SSO login flow of OIDC.

user's operations including authentication, redirection, authorization, and forwarding, are implemented in a software called user agent (i.e., a browser for web applications).

As described above, a "pure" SSO protocol does not include any authentication step, and there is no authentication step between users and an RP. This feature of commonly-used SSO protocols [1–3] brings advantages as below. It enables the IdP to authenticate a user by any appropriate means (e.g., password, two-factor authentication, one-time password, and smart card). Therefore, if a credential was lost or compromised, the user only needs renew it at the IdP; however, if a user needs to prove something to the RP (i.e., some authentication steps are involved), the user has to notify each RP if this secret was leaked (or even the user logins from another computer).

RP Dynamic Registration. In addition to the manual registrations, OIDC also supports dynamic registrations for an RP to register by online means [10]. The (unregistered) RP sends a registration request with endpoints to receive identity tokens (and also other optional information), to the IdP. After a successful registration, the IdP assigns a unique identity to this RP in the response.

2.2 Existing Privacy-Preserving Solutions for SSO

Pairwise pseudonymous identifiers (PPIDs) are recommended by NIST [5] and specified in several SSO protocols [1, 11] to protect user privacy against curious RPs. When issuing an identity token, the IdP encloses a user pseudo-identity (but not the user identity at the IdP) in the token. Given a user, the IdP assigns a unique PPID based on the target RP, so that collusive RPs cannot link the user's different PPIDs across these RPs. PPID-based approaches cannot prevent the IdP-based login tracing, since in the generation of identity tokens the IdP needs to know which RP the user is visiting.

BrowserID [6] and SPRESSO [7] are proposed to defend SSO services against the IdP-based login tracing, but both solutions are instead vulnerable to the threat of RP-based identity linkage. In BrowserID (and its prototypes known as Mozilla Persona [12] and Firefox Accounts [9]), the IdP issues a special "token" to bind the user identity to an ephemeral public key, so that the user uses the corresponding private key to sign a "subsidiary" identity token to bind his identity with the target RP's identity and then sends both tokens to the RP. When a user logins to different RPs, the RPs could still extract the identical user identities from different these tokens and link these login instances. Meanwhile, in SPRESSO an RP creates a one-time but verifiable pseudo-identity for itself in each login instance. Then, the IdP generates an identity token binding this RP pseudo-identity and the user identity. Similarly, the collusive RPs could link a user's multiple login instances based on his permanent user identity in these tokens.

PPIDs cannot be directly integrated in either BrowserID or SPRESSO. PPIDs are assigned in identity tokens based on the visited RP [1, 5, 11], but the IdP receives (*a*) nothing about the visited RP in BrowserID or (*b*) an ephemeral pseudo-identity of the RP in SPRESSO.

2.3 Extended Related Works

Certified RP-Verified Credential. A user of EL PASSO [13] keeps a secret on his device. After authenticating the user, the trusted IdP issues a certified credential binding the secret, also kept on the user's device. When attempting to login to any RP, the user proves that he is the owner of this credential to the RP without exposing the secret; even if such a credential is verified by multiple RPs, user-maintained pseudonyms and anonymous credentials [14] prevent collusive RPs from linking the login instances. UnlimitID [15] presents similar designs, also based on anonymous credentials [14]. PseudoID [16] introduces a token service in addition to the IdP, to blindly sign a pseudonym credential binding the user's secret. Then, the user proves to the RP that he owns this secret to login. These RP-verified credentials protect user privacy well and two kinds of privacy threats are prevented [13, 15, 16], but the user has to by himself locally manage pseudonyms for different RPs. For example, the domain name of an RP is used as a factor to generate the user's account (or pseudonym) at this RP. This bring some burdens to the users, while PPIDs are maintained by the IdP and in UPPRESSO these accounts are determined automatically.

Moreover, in EL PASSO, UnlimitID or PseudoID, a user needs to notify each RP if a credential was lost or compromised, because the user is authenticated by the RP with his credentials to prove that he owns the long-term secret. On the contrary, in the commonly-used SSO systems [1, 2] and privacy-preserving SSO solutions such as BrowserID [6], SPRESSO [7] and also UPPRESSO, the user only needs to renew his credential at the IdP if it was compromised, because (*a*) authentication happens between a user and the IdP and (*b*) an RP verifies only tokens generated by the IdP. Although EL PASSO calls itself an SSO scheme [13] and the service signing tokens or credentials in EL PASSO, UnlimitID and PseudoID is also called the IdP [13, 15, 16], the authentication steps between the user and an RP do not exist in the common SSO login flows.

Formal Analysis on SSO Protocols. Fett et al. [17, 18] formally analyzed OAuth 2.0 and OIDC using an expressive Dolev-Yao style model [19], and presented the attacks of 307 redirection and IdP mix-up. SAML-based SSO is also analyzed [20], and the RP identity is found not to be correctly bound in the identity tokens of a variant designed by Google.

Vulnerable SSO Implementation. Various vulnerabilities were found in SSO implementations for web applications, exploited to launch attacks of impersonation and identity injection by breaking confidentiality [21–25], integrity [21, 25–29] or RP designation [25, 27–30] of identity tokens. In the SSO services of Google and Facebook, logic flaws of the IdPs and

RPs were detected [21]. Integrity of identity tokens was not protected well in several SSO systems [21,26–29] due to software flaws, such as XML signature wrapping [26], incomplete verification by RPs [21,27,29], and IdP spoofing [28,29]. Vulnerabilities break RP designation of identity tokens, due to incorrect binding at the IdP [27,30] and insufficient verification by RPs [28–30].

Automatic tools, such as SSOScan [31], OAuthTester [32] and S3KVetter [30], are designed to detect the violations of confidentiality, integrity, or RP designation of SSO identity tokens. Wang et al. [33] present a tool to detect the vulnerable applications constructed on top of authentication/authorization SDKs, due to the implicit assumptions in these SDKs. Besides, Navas et al. [34] discussed the possible attack patterns of the specification and implementations of OIDC.

In mobile SSO scenarios, the IdP App, IdP-provided SDKs (e.g., an encapsulated WebView) or system browsers are responsible for forwarding identity tokens from the IdP App to RP Apps. However, none of them ensures that the identity tokens are sent to the designated RP only [35,36], because a WebView or the system browser cannot authenticate the RP Apps and the IdP App may be repackaged. The SSO protocols are modified to work for mobile Apps, but these modifications are not well understood by RP developers [35,37]. Vulnerabilities were disclosed in lots of Android applications, to break confidentiality [35–38], integrity [35,37], and RP designation [35,38] of SSO identity tokens. A software flaw was found in Google Apps [23], allowing a malicious RP to hijack a user's authentication attempt and inject a payload to steal the cookie (or identity token) for another RP.

Once a user account at the IdP is compromised, the adversaries will control all his accounts at all RPs. An extension to OIDC, named single sign-off [39], is proposed and then the IdP helps the victim user to revoke all his identity tokens accepted and logout from these RPs.

Anonymous SSO System. Anonymous SSO schemes allow authenticated users to access a service (i.e. RP) protected by the IdP, without revealing their identities. Anonymous SSO was proposed for the global system for mobile (GSM) communications [40], and the notion of anonymous SSO was formalized [41]. Privacy-preserving primitives, such as group signature, zero-knowledge proof, Chebyshev Chaotic Maps and proxy re-verification, etc., were adopted to design anonymous SSO systems [41–44]. Anonymous SSO schemes work for special applications, but are unapplicable to lots of systems that require user identification for customized services.

3 The Identity-Transformation Framework

This section investigates the security and privacy requirements of SSO, and explains the identity dilemma of privacy-preserving SSO. Then, we present the identity-transformation framework which helps to design UPPRESSO.

3.1 Security Requirements of SSO

The primary goal of non-anonymous SSO services is secure user authentication [7], to ensure that a *legitimate* user can always login to an *honest* RP as his permanent identity at this RP, by presenting the *identity tokens* issued by the *honest* IdP.

To achieve this goal, an identity token generated by the IdP is required to specify (*a*) the RP to which the user requests to login (i.e., *RP designation*) and (*b*) exactly the user who is authenticated by the IdP (i.e., *user identification*). Accordingly, an honest RP verifies the designated RP identity (or pseudo-identity) in identity tokens with its own before accepting the tokens; otherwise, a malicious RP could replay a received identity token to the honest RP and login as the victim user. The RP allows the token holder to login as the user identity (or pseudo-identity) specified in the accepted tokens; otherwise, the IdP provides only anonymous services.

The SSO login flow implies *confidentiality* and *integrity* of identity tokens. An identity token shall be forwarded by the authenticated user to the target RP only, not leaked to adversaries; otherwise, an adversary who presents the token, would successfully login to this honest RP. Integrity is necessary, to prevent adversaries from tampering with an identity token, without being detected by the RPs. So identity tokens are signed by the IdP and usually transmitted over HTTPS.

These four security requirements (i.e., RP designation, user identification, confidentiality, and integrity) of SSO identity tokens have been discussed and analyzed [17,18,20], and any vulnerabilities breaking one or more of these properties in SSO systems result in effective attacks [21–32,35–38,45,46]. An adversary might attempt to login to an honest RP as a victim user (i.e., *impersonation*), or allure a victim user to login to an honest RP as the attacker (i.e., *identity injection*). For example, Friendcaster used to accept any received identity token, which violates RP designation [35], so a malicious RP could replay a received identity token to Friendcaster and login as the victim user. The defective IdP of Google ID SSO signs identity tokens where the Email element was not enclosed, while some RPs use a user's Email as his unique username [21]. So this violation of user identity resulted in vulnerable services. Because identity tokens were leaked in different ways [21–25], the eavesdroppers could impersonate the victim users. Some RPs even accept user attributes that are not bound in identity tokens (i.e., a violation of integrity) [21], so that adversaries could insert arbitrary attributes into the identity tokens to impersonate another user at these RPs.

3.2 The Identity Dilemma of Privacy-Preserving SSO

A *completely* privacy-preserving SSO system shall firstly offer the four security properties as mentioned above, while prevent the privacy threats due to the IdP-based login tracing and the RP-based identity linkage. However, to satisfy the requirements of security and privacy at the same time, poses a dilemma in the generation of identity tokens as follows. Table

1 lists the notations used in the following explanation, and the subscript j and/or the superscript i may be omitted sometimes, when there is no ambiguity.

A valid identity token contains the identities (or pseudo-identities) of the authenticated user and the target RP. Since the IdP authenticates users and always knows the user's identity (i.e., ID_U), in order to prevent the IdP-based login tracing, we shall not reveal the target RP's permanent identity (i.e., ID_{RP}) to the IdP in the login flow. So an *ephemeral* pseudo-identity for the RP (i.e., PID_{RP}) shall be used in the communications with the IdP: (*a*) to ensure RP designation, PID_{RP} shall be uniquely associated with the target RP; and (*b*) the IdP cannot derive any information about ID_{RP} from any PID^i_{RP}, which implies PID^i_{RP} in multiple login instances shall be independent of each other.[1]

Next, in order to prevent the RP-based identity linkage, the IdP shall not directly enclose ID_U in identity tokens. A pseudo-identity for the user (i.e., PID_U) shall be bound instead: (*a*) the RP cannot derive any information about ID_U from any $PID_{U,j}$, which implies $PID_{U,j}$ for different RPs shall be independent of each other; (*b*) in multiple login instances to the RP, PID^i_U shall be independent of each other or generated ephemerally, to prevent the IdP-based login tracing;[2] and (*c*) to ensure user identification, an *ephemeral* PID^i_U in each login instance shall facilitate the RP to correlate it with the *permanent* account (i.e., $Acct$) at this RP.

We summary these identities and pseudo-identities. That is, (*a*) an identity tokens shall contain only the pseudo-identities, i.e., $PID^i_{U,j}$ and $PID^i_{RP_j}$, which are independent of each other for different RPs and in multiple login instances, respectively, and (*b*) these two *ephemeral* pseudo-identities enable the RP to calculate the *permanent* account, i.e., $Acct_j$.

We illustrate the relationships among the identities and pseudo-identities in identity tokens in Figure 2. The red and green blocks represent permanent identities and ephemeral pseudo-identities, respectively. The arrows denote how the pseudo-identities are transformed. The figure describes the *identity dilemma* of privacy-preserving SSO:

> Given an authenticated user and an unknown RP (i.e., permanent ID_U and ephemeral PID_{RP}), the IdP is expected to generate an ephemeral pseudo-identity (i.e., PID_U) which will be correlated with the user's permanent identity at this RP (i.e., $Acct$), with knowing nothing about the RP's identity or the user's account at this RP (i.e., ID_{RP} or $Acct$).

[1] Even when the target RP is kept unknown to the IdP, the IdP shall not link multiple login instances which attempt to visit this RP.

[2] If PID^i_U is not completely independent of each other, it implies the IdP could link multiple login instances which attempt to visit this RP.

Table 1: The (pseudo-)identities in privacy-preserving SSO.

Notation	Description	Attribute
ID_U	The user's unique identity at the IdP.	Permanent
ID_{RP_j}	The j-th RP's unique identity at the IdP.	Permanent
$PID^i_{U,j}$	The user's pseudo-identity, in the user's i-th login instance to the j-th RP.	Ephemeral
$PID^i_{RP_j}$	The j-th RP's pseudo-identity, in the user's i-th login instance to this RP.	Ephemeral
$Acct_j$	The user's identity (or account) at the j-th RP.	Permanent

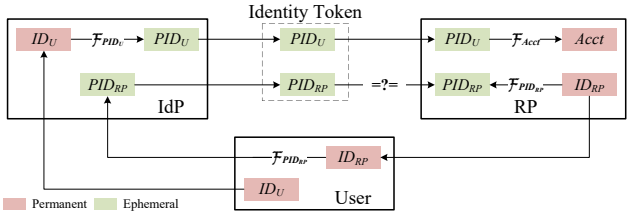

Figure 2: Identity transformations in privacy-preserving SSO.

We explicitly distinguish a user's identity at the RP, i.e., the account, from (a) *the user's identity at the IdP and* (b) *the user's pseudo-identity in identity tokens.* This conceptualization is proposed for the first time, to the best of our knowledge, and it essentially helps to effectively build the identity-transformation framework.

3.3 Identity Transformation

The privacy-protection problem of SSO is converted into an identity-transformation challenge, to design three *identity-transformation functions* as follows.

- $\mathcal{F}_{PID_{RP}}(ID_{RP}) = PID_{RP}$, calculated by the user and/or the RP. From the IdP's view, $\mathcal{F}_{PID_{RP}}()$ is a one-way function and the calculated PID_{RP} appears a random variable.
- $\mathcal{F}_{PID_U}(ID_U, PID_{RP}) = PID_U$, calculated by the IdP. From the RP's view, $\mathcal{F}_{PID_U}()$ is a one-way function and the calculated PID_U appears a random variable.
- $\mathcal{F}_{Acct}(PID_U, PID_{RP}) = Acct$, calculated by the RP. Given ID_U and ID_{RP}, $Acct$ keeps unchanged; i.e., in the user's any i-th and i'-th ($i \neq i'$) login instances to the RP, $\mathcal{F}_{Acct}(PID^i_U, PID^i_{RP}) = \mathcal{F}_{Acct}(PID^{i'}_U, PID^{i'}_{RP})$.

In an SSO login flow with identity-transformation functions, a user firstly negotiates an ephemeral PID_{RP} with the target RP. Then, an identity-token request with PID_{RP} is sent by the user to the IdP. After authenticating the user as ID_U, the IdP calculates an ephemeral PID_U based on ID_U and the received PID_{RP}, and issues an identity token binding PID_U and PID_{RP}. This token is forwarded by the user to the RP. Finally, after verifying the designated RP pseudo-identity in the token, the RP calculates $Acct$ and allows the token holder to login as $Acct$.

The identity-transformation functions will satisfy the privacy requirements, while the authentication steps are kept independent of the SSO protocol. In particular, as shown in

Figure 1, the authentication steps are conducted only between a user and the IdP, and these steps do not deal with identity tokens. After the identity-transformation functions are integrated, the steps dealing with identity tokens do not involve any user credentials, as those in the commonly-used SSO protocols [1–3].

4 The Designs of UPPRESSO

This section presents the threat model and assumptions of UPPRESSO. Then, we design three identity-transformation functions satisfying the privacy requirements, and present the detailed protocols for web applications. The compatibility with OIDC is also discussed, which will help the adoption and deployment of UPPRESSO.

4.1 Threat Model

The IdP is curious-but-honest, while some users and RPs could be compromised. Malicious users and RPs behave arbitrarily and might collude with each other, attempting to break the security and privacy guarantees for benign users.

Curious-but-honest IdP. The IdP strictly follows the protocol, while being interested in learning user privacy. For example, it might store all received messages to infer the relationship among ID_U, ID_{RP}, PID_U, and PID_{RP} to track a user's login activities. We assume the IdP is well-protected. For example, the IdP is trusted to maintain the private key for signing identity tokens and RP certificates. So, adversaries cannot forge such tokens or certificates. We do not consider the collusion of the IdP and RPs in this paper. In fact, if the IdP could collude with an RP, a user would finish a login instance completely with collusive entities and it is rather impossible to prevent the IdP-based login tracing.

Malicious Users. We assume the adversary could control a set of users, by stealing users' credentials or registering Sybil accounts in the system. They want to impersonate a victim user at honest RPs, or allure the user to login to an honest RP under the adversary's account. A malicious user might modify, insert, drop or replay a message, or behave arbitrarily in any SSO login instances.

Malicious RPs. The adversary could also control a set of RPs, by registering at the IdP as an RP or exploiting software vulnerabilities to compromise some RPs. The malicious RPs might behave arbitrarily to break the security and privacy guarantees of UPPRESSO. For example, a malicious RP might manipulate PID_{RP} in a login instance, attempting to allure honest users to return an identity token which might be accepted by an honest RP; or, it might manipulate PID_{RP} to affect the generation of PID_U, attempting to analyze the relationship between ID_U and PID_U.

Collusive Users and RPs. Malicious users and RPs might collude with each other. For example, an adversary could first pretend to be an RP and allure victim users to forward an identity token to him. With this identity token, it could try to impersonate the victim user and login to some honest RP.

Table 2: The notations in the UPPRESSO protocols.

Notation	Description
$\mathbb{E}$	An elliptic curve over a finite field $\mathbb{F}_q$, where the ECDLP is computationally impossible.
G, n	G is a base point (or generator) of $\mathbb{E}$, and the order of G is a prime number n.
ID_U	$ID_U = u$, $1 < u < n$; the user's unique identity at the IdP.
ID_{RP_j}	$ID_{RP} = [r]G$, $1 < r < n$; the j-th RP's unique identity at the IdP.
t	The user-generated random number in a login instance, $1 < t < n$.
$PID^i_{RP_j}$	$PID_{RP} = [t]ID_{RP} = [tr]G$; the j-th RP's pseudo-identity, in the user's i-th login instance to this RP.
$PID^i_{U,j}$	$PID_U = [ID_U]PID_{RP} = [utr]G$; the user's pseudo-identity, in the user's i-th login instance to the j-th RP.
$Acct_j$	$Acct = [t^{-1}]PID_U = [ID_U]ID_{RP} = [ur]G$; the user's account at the j-th RP.
SK, PK	The IdP's key pair, a private key and a public key, to sign and verify identity tokens and RP certificates.
$Enpt_{RP_j}$	The j-th RP's endpoint, to receive the identity tokens.
$Cert_{RP_j}$	The RP certificate signed by the IdP, binding ID_{RP_j} and $Enpt_{RP_j}$.
$PEnpt^i_{U,j}$	A user-generated random "pseudo-endpoint", in the user's i-th login instance to the j-th RP.

4.2 Assumptions

We assume that a user never authorizes the IdP to enclose any *distinctive attributes* in identity tokens, where distinctive attributes are identifiable information such as telephone number, driver license, Email, etc. The user does not register distinctive attributes at any RP, either. Therefore, the privacy leakage due to user re-identification by distinctive attributes is out of the scope of this work.

HTTPS is adopted to secure the communications between honest entities, and the adopted cryptographic primitives are secure. The software stack of honest entities is correctly implemented, so it transmits messages to the receivers as expected.

We focus on the privacy attacks introduced by the design of SSO protocols, but not network attacks such as the traffic analysis that traces a user's login activities from network packets. Such attacks shall be prevented by other defenses.

4.3 Identity-Transformation Functions

We design three identity-transformation functions, $\mathcal{F}_{PID_{RP}}$, $\mathcal{F}_{PID_U}$ and $\mathcal{F}_{Acct}$, over an elliptic curve $\mathbb{E}$, where G is a base point (or generator) of this elliptic curve and the order of G is a big prime number denoted as n. Table 2 lists the notations, and the subscript j and/or the superscript i may be omitted in the case of no ambiguity.

ID_U is a unique integer satisfying $1 < ID_U < n$, and ID_{RP} is a unique point on $\mathbb{E}$. When a user is registering, a unique

random number u $(1 < u < n)$ is generated and $ID_U = u$ is assigned to this user; when an RP is initially registering, a unique random number r $(1 < r < n)$ is generated by the IdP and $ID_{RP} = [r]G$ is assigned to this RP. Here, $[r]G$ is the addition of G on the curve r times.

***ID_{RP}-PID_{RP}* Transformation.** In each login instance, the user selects a random number t $(1 < t < n)$ as the trapdoor and calculates PID_{RP} as below.

$$PID_{RP} = \mathcal{F}_{PID_{RP}}(ID_{RP}) = [t]ID_{RP} = [tr]G \quad (1)$$

***ID_U-PID_U* Transformation.** On receiving an identity-token request with ID_U and PID_{RP}, the IdP calculates PID_U.

$$PID_U = \mathcal{F}_{PID_U}(ID_U, PID_{RP}) = [ID_U]PID_{RP} = [utr]G \quad (2)$$

***PID_U-Acct* Transformation.** In the negotiation of PID_{RP}, the user sends the trapdoor t to the target RP. So the RP also calculates PID_{RP} to verify the designated RP pseudo-identity in identity tokens. After verifying an identity token binding PID_U and PID_{RP}, the RP calculates $Acct$ as below.

$$Acct = \mathcal{F}_{Acct}(PID_U, PID_{RP}) = [t^{-1} \bmod n]PID_U \quad (3)$$

From Equations 1, 2 and 3, it is derived that

$$Acct = [t^{-1}utr \bmod n]G = [ur]G = [ID_U]ID_{RP}$$

The RP obtains the identical permanent account from different identity tokens in multiple login instances, with the help of t from the user. Given a user, the accounts at different RPs are inherently unique. Moreover, (a) due to the elliptic curve discrete logarithm problem (ECDLP), it is computationally infeasible for the RP to derive ID_U from either PID_U or $Acct$; and (b) because t is a random number kept secret to the IdP, it is impossible for the IdP to derive ID_{RP} from PID_{RP}.

4.4 The Design Specific for Web Applications

We further propose the designs specific for web applications, and these designs enable UPPRESSO to provide SSO services for users with standard browsers. More efficient but less portable implementations with browser extensions are discussed in Section 7.

First of all, in the SSO login flow, the user has to deal with RP endpoints (i.e., the URLs to receive identity tokens) by himself. In existing SSO protocols, an RP initially registers its endpoint at the IdP, and then in each login instance, the IdP will set the endpoint in the identity-token response. This instructs the browser to forward it correctly; otherwise, confidentiality of identity tokens might be broken.

Because in UPPRESSO the IdP is not aware of the visited RP and cannot set the RP endpoints, *RP certificates* are designed to instructs the user agents (or browsers) about RP endpoints. An RP certificate is a document signed by the IdP in the RP registration, binding the RP's identity and its endpoint. This certificate is sent by the RP in the login flow, so a user is able to forward the identity tokens to this verified endpoint.

Secondly, browser scripts are needed to implement the functions by users, including the generation of t and PID_{RP} and the dealing with RP certificates and endpoints, for they are not standard functions of a browser. Two scripts, one downloaded from the IdP and the other from the target RP, work together (with the standard browser functions) as the user agent of UPPRESSO. The RP script maintains the communications with the RP, and it does not communicates directly with the IdP because an HTTP request launched by the RP script will automatically carry an HTTP `Referer` header, which discloses the RP's domain. The IdP script downloaded from the IdP, is responsible for the communications with the IdP, and two scripts communicates with each other through the `postMessage` HTML5 API.

Finally, the IdP's public key is downloaded in the IdP script to verify RP certificates. So the user agent does not configure anything locally, as it does in most existing SSO systems.

4.5 The UPPRESSO Protocols

System Initialization. The IdP generates a key pair (SK, PK) to sign/verify identity tokens and RP certificates. Then, the IdP keeps SK secret, while PK is publicly known.

RP Initial Registration. Each RP launches an initial registration operation to finish configurations. In particular, an RP registers itself at the IdP to obtain ID_{RP} and the corresponding RP certificate $Cert_{RP}$ as follows:

1. The RP sends a registration request to the IdP, including the endpoint to receive identity tokens, and other optional information.
2. The IdP generates a unique random number r, calculates $ID_{RP} = [r]G$, and assigns ID_{RP} to this RP. The IdP then signs $Cert_{RP} = [ID_{RP}, Enpt_{RP}, *]_{SK}$, where $[\cdot]_{SK}$ means a message signed using SK and $*$ denotes supplementary information such as the RP's common name and Email, and returns $Cert_{RP}$ to the RP.
3. The RP verifies $Cert_{RP}$ using PK, and accepts ID_{RP} and $Cert_{RP}$ if they are valid.

User Registration. UPPRESSO adopts a similar user registration operation as the ones in existing SSO systems. Each user registers once at the IdP to set up a unique user identity ID_U and the corresponding credential.

SSO Login. An SSO login instance is typically launched through a browser, when a user requests to login to an RP. It consists of five steps, namely script downloading, RP identity transformation, PID_{RP} registration, identity-token generation, and $Acct$ calculation, as shown in Figure 3. In this figure, the operations by the IdP are linked by a vertical line, so are the RP's operations. Two vertical lines split the user's operations

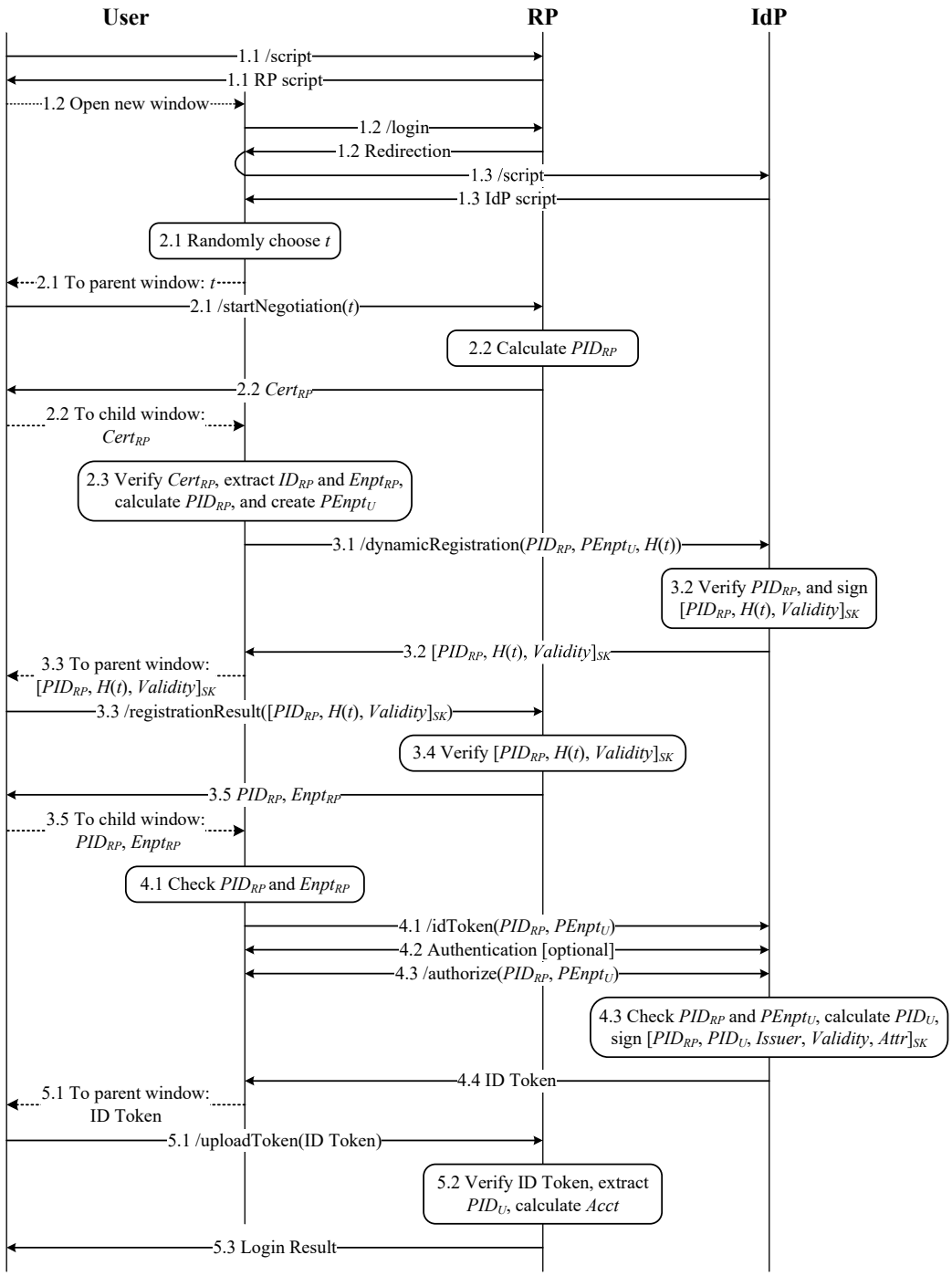

Figure 3: The SSO login flow of UPPRESSO.

into two groups (i.e., in two browser windows), one of which is to communicate with the IdP, and the other with the target RP. Each solid horizontal line means some messages between the user and the IdP (or the RP), and each dotted line means a `postMessage` invocation between two scripts (or windows) within the browser.

1. *Script Downloading.* The browser downloads the scripts from the IdP and the visited RP.

1.1 When attempting to visit any protected resources at the RP, the user downloads the RP script.

1.2 The RP script opens a window in the browser to visit the login path at the RP, which is then redirected to the IdP.

1.3 The redirection to the IdP downloads the IdP script.

2. RP Identity Transformation. The user and the RP negotiate $PID_{RP} = [t]ID_{RP}$.

2.1 The IdP script in the browser chooses a random number t ($1 < t < n$) and sends it to the RP script through `postMessage`. Then, the RP script sends t to the RP.

2.2 On receiving t, the RP verifies $1 < t < n$ and calculates PID_{RP}. The RP replies with $Cert_{RP}$, which is then transmitted from the RP script to the IdP script.

2.3 The IdP script verifies $Cert_{RP}$, extracts ID_{RP} and $Enpt_{RP}$ from $Cert_{RP}$ and calculates $PID_{RP} = [t]ID_{RP}$. It then creates a random endpoint $PEnpt_U$ for this login instance, to receive identity tokens from the IdP.

3. PID_{RP} Registration. The user dynamically registers an ephemeral PID_{RP} at the IdP.

3.1 The IdP script sends the PID_{RP}-registration request $[PID_{RP}, PEnpt_U, H(t)]$ to the IdP, where $H()$ is a collision-free one-way hash function.

3.2 The IdP checks the list of unexpired PID_{RP} to verify the received PID_{RP} is a unique point on $\mathbb{E}$ among them. Then, it signs the response $[PID_{RP}, H(t), Validity]_{SK}$, where $Validity$ indicates when PID_{RP} will expire (typically, in 3 to 5 minutes). The IdP maintains the list of unexpired $[PID_{RP}, PEnpt_U, H(t), Validity]$ and periodically deletes expired ones from it.

3.3 The IdP script forwards the PID_{RP}-registration result to the RP through the RP script.

3.4 The RP verifies the IdP's signature, and accepts the registration result only if PID_{RP} and $H(t)$ match those in the negotiation and it does not expire.

3.5 The RP constructs an identity-token request with PID_{RP} and $Enpt_{RP}$, which is then forwarded to the IdP script through the RP script.

The PID_{RP}-registration result signed by the IdP ensures that PID_{RP} is *unique* within its validity; otherwise, RP designation is broken. $H(t)$ is a *necessary* nonce to distinguish different login instances, because there is a very small probability that an identical PID_{RP} is calculated for two RPs from two different pairs of ID_{RP} and t (i.e., $[t]ID_{RP_j} = [tr]G = [t'r']G = [t']ID_{RP_{j'}}$);[3] otherwise, although PID_{RP} is unique within its validity from the IdP's view, multiple RPs still share an identical PID_{RP}. This nonce prevents one PID_{RP}-registration result (and the subsequent identity token) from being accepted by different RPs.

4. Identity-Token Generation. The IdP calculates $PID_U = [ID_U]PID_{RP}$ and signs the identity token.

4.1 The IdP script checks that PID_{RP} is the one registered in Step 3.1 and $Enpt_{RP}$ matches the one in $Cert_{RP}$. Then, the IdP script replaces $Enpt_{RP}$ with $PEnpt_U$ in the identity-token request and sends this modified request to the IdP.

4.2 On receiving an identity-token request, the IdP authenticates the user if he has not been authenticated yet.

4.3 After obtaining the user's authorization to enclose the requested attributes, the IdP checks whether the received pair of PID_{RP} and $PEnpt_U$ is in the list of unexpired PID_{RP} or not, and calculates $PID_U = [ID_U]PID_{RP}$ for the authenticated user. The IdP then signs an identity token $[PID_{RP}, PID_U, Issuer, Validity, Attr]_{SK}$, where $Issuer$ is the IdP's identity, $Validity$ is the validity period, and $Attr$ contains the requested attributes.

4.4 The IdP sends the identity token to $PEnpt_U$.

5. Acct Calculation. The RP verifies the identity token and allows the user to login.

5.1 The IdP script forwards this token to the RP script, which then sends it to the RP through $Enpt_{RP}$.

5.2 The RP verifies the identity token, including the IdP's signature and its validity period. It also verifies PID_{RP} in the token is consistent with the one negotiated in Step 2.2. Then, the RP extracts PID_U, and calculates $Acct = [t^{-1}]PID_U$.

5.3 The RP returns the login result, and allows the user to login as $Acct$.

If any verification or check fails in some step, the flow will be halted immediately. For example, the user halts the flow if $Cert_{RP}$ is invalid or PID_{RP} in the identity-token request is inconsistent with the negotiated one. The IdP rejects a PID_{RP}-registration request, if there is any unexpired but identical PID_{RP} in the list, and the RP rejects identity tokens until it accepts a PID_{RP}-registration result. The IdP rejects an identity-token request, if the pair of PID_{RP} and $PEnpt_U$ is not in the unexpired list. Or, the RP rejects an identity token if PID_{RP} in the token does not match the negotiated one.

4.6 Compatibility with OIDC

Among the five steps of the SSO login flow in UPPRESSO, the script downloading prepares the user agent. The user agent of SSO deals with the communications between the IdP and the RP, which are redirected by browsers in OIDC. On the

[3] Although we assume $H()$ is collision-free, commonly-used one-way hash functions such as SHA-1 and SHA-256 are fine, because it is impossible that the following equations hold simultaneously: $r \neq r'$, $[tr]G = [t'r']G$, and $H(t) = H(t')$.

other hand, in UPPRESSO when sending the identity-token request, the script replaces $Enpt_{RP}$ with $PEnpt_U$, and then the script forwards the identity token to $Enpt_{RP}$ which is extracted from the RP certificate.

In the step of RP identity transformation, most operations are conducted by the user agent, while the RP only receives t to calculate PID_{RP} and sends $Cert_{RP}$. The operations in the PID_{RP} registration are almost identical to those in the RP Dynamic Registration of OIDC [10], except that in OIDC the IdP assigns the RP's identity while in UPPRESSO this (pseudo-)identity is generated by the registered entity.

The operations in the steps of identity-token generation and $Acct$ calculation, are actually identical to those in the implicit SSO login flow of OIDC [1], while (a) the calculation of PID_U is viewed as a method to generate PPIDs by the IdP and (b) the calculation of $Acct$ is viewed as a mapping from the user identity in tokens to the local account at the RP.

5 The Analysis of Security and Privacy

In this section, we presents the analysis that UPPRESSO achieves the required properties of security and privacy.

5.1 Security

UPPRESSO satisfies the four security requirements of identity tokens in SSO services, as listed in Section 3.1.

RP Designation. An identity token binding PID_U and PID_{RP}, designates the target RP, and only the target RP. An honest RP calculates PID_{RP} by itself with the trapdoor t sent from the user, and checks PID_{RP} in the PID_{RP}-registration result and the identity token. So the target RP will accept this token.

Meanwhile, the honest IdP guarantees that, within its validity period, the PID_{RP} will be registered only once. An honest RP is ready to accept an identity token binding PID_{RP}, only after it receives the signed PID_{RP}-registration result. Because both PID_{RP} and $H(t)$ in the registration result are checked by the RP and then the registration result $[PID_{RP}, H(t), Validity]_{SK}$ is acceptable to only one honest RP, the identity token designates only one RP.

User Identification. An honest RP always derives an identical permanent account from different identity tokens binding PID_U and PID_{RP}. That is, in the user's any i-th and i'-th ($i \neq i'$) login instances to the RP, $\mathcal{F}_{Acct}(PID_U^i, PID_{RP}^i) = \mathcal{F}_{Acct}(PID_U^{i'}, PID_{RP}^{i'}) = [ID_U]ID_{RP}$.

In the calculation of $Acct = [t^{-1}]PID_U = [t^{-1}][u]PID_{RP}$, t and PID_{RP} are checked by the honest RP in the PID_{RP}-registration result, and PID_U is calculated by the IdP based on (a) the authenticated user, i.e., $ID_U = u$, and (b) the registered PID_{RP}. Thus, the calculated account is always exactly the authenticated user's account at the RP (i.e., $[ID_U]ID_{RP}$). For example, two malicious users, whose identities are $ID_U = u$ and $ID_{U'} = u'$, could attempt to login to RP_j and $RP_{j'}$, respectively. If the generated t and t' happen to satisfy that $PID_{RP} = [t]ID_{RP_j} = [tr]G = [t'r']G = [t']ID_{RP_{j'}}$, these collusive users could arbitrarily choose to register either $[PID_{RP}, PEnpt_U, H(t)]$ or $[PID_{RP}, PEnpt_{U'}, H(t')]$ at the IdP, to receive an identity token binding either $PID_U = [u]PID_{RP}$ or $PID_{U'} = [u']PID_{RP}$ (and also PID_{RP}).[4] However, when such a token is signed for RP_j, the calculated $Acct$ is $[ur]G$ or $[u'r't't^{-1}]G = [u'r]G$; when it is signed for $RP_{j'}$, $[urtt'^{-1}]G = [ur']G$ or $[u'r']G$ is calculated. That is, even in this collusive case, the calculated account is still the authenticated user's account at the RP, and it does not result in any attack.

Confidentiality. There is no event leaking the identity tokens to any malicious entity other than the authenticated user and the designated RP. First of all, the communications among the IdP, RPs and users, are protected by HTTPS, and the `postMessage` HTML5 API ensures the dedicated channels between two scripts within the browser, so that adversaries cannot eavesdrop the identity tokens. Meanwhile, the honest IdP sends the identity token only to the authenticated user, and this user forwards it to the RP through $Enpt_{RP}$. The binding of $Enpt_{RP}$ and ID_{RP} is ensured by the signed RP certificate, so only the designated target RP receives this identity token.

Integrity. The identity token binds ID_U and ID_{RP} implicitly or explicitly, and any breaking will result in some failed check or verification in the login flow. The integrity is ensured by the IdP's signatures: (a) the identity token binding PID_U and PID_{RP}, is signed by the IdP, and (b) the relationship between PID_{RP} and t (or collision-free $H(t)$) is also bound in the PID_{RP}-registration result. Thus, ID_U and ID_{RP} are actually bound by the IdP's signatures, due to the one-to-one mapping between (a) the pair of ID_U and ID_{RP} and (b) the triad of PID_U, PID_{RP}, and t.

We also formally analyze the security properties of UPPRESSO, based on an Dolev-Yao style model of the web infrastructure [7], which has been used in the formal analysis of SSO protocols such as OAuth 2.0 [17] and OIDC [18]. The Dolev-Yao style model abstracts the entities in a web system, such as browsers and web servers, as *atomic processes*, which communicate with each other through *events*. It also defines *script processes* to formulate client-side scripts, i.e., JavaScript code, so a web system consists of a set of atomic and script processes.

The UPPRESSO system contains an IdP process, a finite set of web servers for honest RPs, a finite set of honest browsers, and a finite set of attacker processes. Here, we consider all RP processes and browser processes are honest, while model an RP or a browser controlled by an adversary as atomic attacker processes. It also contains script_rp, script_idp and script_attacker, where script_rp and script_idp are honest scripts downloaded from an RP process and the IdP process,

[4] Such a token designates either RP_j or $RP_{j'}$, but only one RP because there is only one acceptable PID_{RP}-registration result which is signed by the IdP. So RP designation is not violated in this case.

respectively, and script_attacker denotes a script downloaded by an attacker process that exists in all browser processes.

After formulating UPPRESSO by the Dolev-Yao style model, we trace the identity token, starting when it is generated and ending when it is consumed, to ensure that an identity token is not stolen or tempered by adversaries. For example, we locate the generation of an identity token in UPPRESSO, and trace back to the sources where PID_U, PID_{RP} and other values enclosed in this token are generated and transmitted, to ensure that no adversary able to retrieve or manipulate them. The tracing of identity tokens also confirm no adversary retrieves the token.

Finally, we formally prove that, *user identification*, *RP designation*, *confidentiality*, and *integrity* are fulfilled in UPPRESSO. The details on the Dolev-Yao web model and the security proof of UPPRESSO are in the appendix.

5.2 Privacy

We show that UPPRESSO effectively prevents the attacks of IdP-based login tracing and RP-based identity linkage.

IdP-based Login Tracing. The information accessible to the IdP and derived from the RP's identity, is only PID_{RP}, where $PID_{RP} = [t]ID_{RP}$ is calculated by the user. Because (*a*) t is a number randomly chosen from $(1,n)$ by the user and kept secret to the IdP and (*b*) $ID_{RP} = [r]G$ and G is the base point (or generator) of $\mathbb{E}$, the IdP has to view PID_{RP} as randomly and independently chosen from $\mathbb{E}$, and cannot distinguish $[t]ID_{RP_j} = [tr]G$ from $[t']ID_{RP_{j'}} = [t'r']G$. So, the IdP cannot derive the RP's identity or link any pair of PID^{i}_{RP} and $PID^{i'}_{RP}$, and then the IdP-based identity linkage is impossible.

RP-based Identity Linkage. We prove UPPRESSO prevents the RP-based identity linkage, based on the elliptic curve decision Diffie-Hellman (ECDDH) assumption [47].

Let $\mathbb{E}$ be an elliptic curve over a finite field $\mathbb{F}_q$, and P be a point on $\mathbb{E}$ of order n. For any probabilistic polynomial time (PPT) algorithm $\mathcal{D}$, ($[x]P$, $[y]P$, $[xy]P$) and ($[x]P$, $[y]P$, $[z]P$) are computationally indistinguishable, where x, y and z are integer numbers randomly and independently chosen from $(1,n)$. Let $Pr\{\}$ denote the probability and we define

$$\begin{aligned} Pr_1 =& Pr\{\mathcal{D}(P,[x]P,[y]P,[xy]P) = 1\} \\ Pr_2 =& Pr\{\mathcal{D}(P,[x]P,[y]P,[z]P) = 1\} \\ \varepsilon(k) =& Pr_1 - Pr_2 \end{aligned}$$

Then, $\varepsilon(k)$ becomes negligible with the security parameter k.

In the login flow, an RP holds ID_{RP} and $Acct$, receives t, calculates PID_{RP}, and verifies two signed messages (i.e., PID_{RP} and $H(t)$ in the PID_{RP}-registration result, and PID_{RP} and PID_U in the identity token). After filtering out the redundant information (i.e., $PID_{RP} = [t]ID_{RP}$ and $Acct = [t^{-1}]PID_U$), the RP actually receives only (ID_{RP}, t, PID_U) in each SSO login instance, where $PID_U = [ID_U][t]ID_{RP}$.

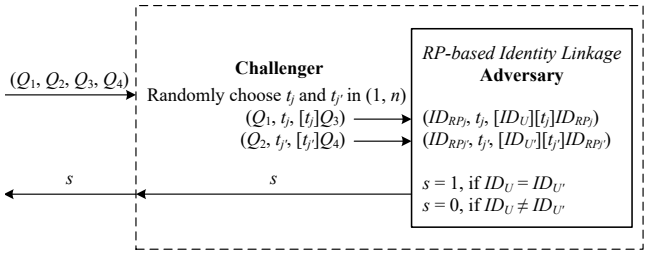

Figure 4: The algorithm based on the RP-based identity linkage, to solve the ECDDH problem.

In the RP-based identity linkage, two RPs bring two triads received in SSO login instances, $(ID_{RP_j}, t_j, [ID_U][t_j]ID_{RP_j})$ and $(ID_{RP_{j'}}, t_{j'}, [ID_{U'}][t_{j'}]ID_{RP_{j'}})$. We describe the attack as the following game $\mathcal{G}$ between an adversary and a challenger: the adversary receives $(ID_{RP_j}, t_j, [ID_U][t_j]ID_{RP_j}, ID_{RP_{j'}}, t_{j'}, [ID_{U'}][t_{j'}]ID_{RP_{j'}})$ from the challenger, and outputs the result s. The result is 1, when the adversary guesses that $ID_U = ID_{U'}$; otherwise, the adversary thinks they are different users (i.e., $ID_U \neq ID_{U'}$) and $s = 0$.

We define Pr_c as the probability that the adversary outputs $s = 1$ when $ID_U = ID_{U'}$ (i.e., a *correct* identity linkage), and $Pr_{\bar{c}}$ as the probability that $s = 1$ but $ID_U \neq ID_{U'}$ (i.e., an *incorrect* result). The successful RP-based identity linkage means the adversary has non-negligible advantages in $\mathcal{G}$.

We design a PPT algorithm $\mathcal{D}^*$ based on $\mathcal{G}$, shown in Figure 4. The input of $\mathcal{D}^*$ is in the form of (Q_1, Q_2, Q_3, Q_4), and each Q_i is a point on $\mathbb{E}$. On receiving the input, the challenger of $\mathcal{G}$ randomly chooses t_j and $t_{j'}$ in $(1,n)$, and sends $(Q_1, t_j, [t_j]Q_3, Q_2, t_{j'}, [t_{j'}]Q_4)$ to the adversary. Finally, it directly outputs s from the adversary in $\mathcal{G}$ as the result of $\mathcal{D}^*$.

Let $(P, [x]P, [y]P, [xy]P)$ and $(P, [x]P, [y]P, [z]P)$ be two inputs of $\mathcal{D}^*$. Thus, we obtain

$$\begin{aligned} & Pr\{\mathcal{D}^*(P,[x]P,[y]P,[xy]P) = 1\} \\ =& Pr\{\mathcal{G}(P,t_j,[t_j][y]P,[x]P,t_{j'},[t_{j'}][xy]P) = 1\} \\ =& Pr\{\mathcal{G}(P,t_j,[y][t_j]P,[x]P,t_{j'},[y][t_{j'}][x]P) = 1\} = Pr_c \end{aligned} \tag{4}$$

$$\begin{aligned} & Pr\{\mathcal{D}^*(P,[x]P,[y]P,[z]P) = 1\} \\ =& Pr\{\mathcal{G}(P,t_j,[t_j][y]P,[x]P,t_{j'},[t_{j'}][z/x][x]P) = 1\} \\ =& Pr\{\mathcal{G}(P,t_j,[y][t_j]P,[x]P,t_{j'},[z/x][t_{j'}][x]P) = 1\} = Pr_{\bar{c}} \end{aligned} \tag{5}$$

Equation 4 is equal to Pr_c because it represents the correct case of $ID_U = ID_{U'} = y$, while Equation 5 is $Pr_{\bar{c}}$ for it represents the incorrect case of $ID_U = y$ but $ID_{U'} = z/x \bmod n$.

The adversary has non-negligible advantages in $\mathcal{G}$ means $Pr_c - Pr_{\bar{c}} > \sigma(k)$, and then $\mathcal{D}^*$ significantly distinguishes (P, $[x]P$, $[y]P$, $[xy]P$) from (P, $[x]P$, $[y]P$, $[z]P$), which violates the ECDDH assumption. So the adversary has no advantages in the game, and the RP-based identity linkage is computationally impossible.

6 Implementation and Evaluation

We have implemented the UPPRESSO prototype system, and evaluated its performance by comparing it with (*a*) the PPID-enhanced OIDC protocol [48] which only prevents the RP-based identity linkage and (*b*) SPRESSO [7] which only prevents the IdP-based login tracing.

6.1 Prototype Implementation

First of all, three identity-transformation functions are defined over the NIST P256 elliptic curve. RSA-2048 and SHA-256 are adopted as the signature algorithm and the hash function, respectively.

The IdP is built on top of MITREid Connect [48], an open-source OIDC Java implementation, and only small modifications are needed as follows. We add only 3 lines of Java code to calculate PID_U, about 20 lines to modify the way to send identity tokens, and about 50 lines to the function of RP Dynamic Registration to support the step of PID_{RP} registration (i.e., checking PID_{RP} and signing the registration result). The calculations of ID_{RP}, PID_U, and RSA signature are implemented based on Java built-in cryptographic libraries.

The user-side functions are implemented by scripts from the IdP and RPs, containing about 200 lines and 150 lines of JavaScript code, respectively. The cryptographic computations, e.g., $Cert_{RP}$ verification and PID_{RP} negotiation, are implemented based on jsrsasign [49], an efficient JavaScript cryptographic library.

We also provide a Java SDK for RPs in UPPRESSO. The SDK provides two functions to encapsulate the protocol steps: one to request identity tokens, and the other to derive the accounts from identity tokens. The SDK is implemented based on the Spring Boot framework with about 1,000 lines of Java code and cryptographic computations are implemented based on the Spring Security library. An RP only needs to invoke these two SDK functions for the integration, by less than 10 lines of code.

6.2 Performance Evaluation

Three machines connected in an isolated 1Gbps LAN, build the experimental SSO environment. The CPUs are Intel Core i7-4770 3.4 GHz for the IdP, Intel Core i7-4770S 3.1 GHz for the RP, and Intel Core i5-4210H 2.9 GHz for users. Each machine is configured with 8 GB RAM and installs Windows 10 as the operating system. The user agent is Chrome v75.0.3770.100.

We compared UPPRESSO with MITREid Connect and SPRESSO. In the evaluation, MITREid Connect runs with the standard implicit login flow of OIDC, while the identity tokens in SPRESSO are also forwarded by a user to the RP, similarly to the implicit login flow of OIDC to some extent. In the identity tokens of SPRESSO, PID_{RP} is the encrypted RP domain, while the one-time symmetric key only known by the RP and the user. They also configure RSA-2048 and SHA-256 in the generation of identity tokens.

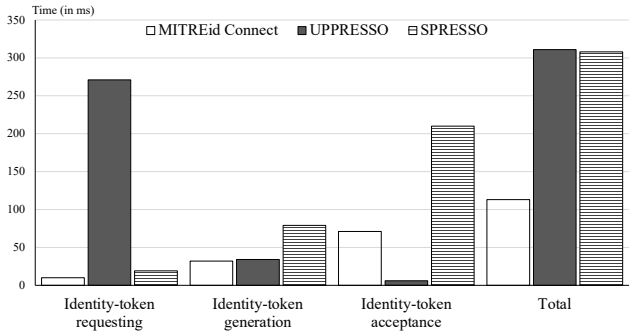

Figure 5: The time cost of SSO login flow.

MITREid Connect provides open-source Java implementations of IdP and RP SDK, while SPRESSO implements all entities by JavaScript based on node.js. We then implemented the RPs based on Spring Boot for UPPRESSO and MITREid Connect, by integrating the corresponding SDK. The RPs in three schemes provide the same function, i.e., simply extract the user's account from verified identity tokens.

We measured the time for a user to login to an RP and calculated the average value of 1,000 measurements. We divide a login flow into 3 parts: *Preparation and identity-token requesting* (for UPPRESSO, it includes Steps 1-3 in Figure 3), as the RP constructs an identity-token request transmitted to the IdP, cooperatively with the user sometimes; *Identity-token generation* (Step 4), when the IdP generates an identity token (but the user authentication is not included); and *Identity-token acceptance* (Step 5 in Figure 3), as the RP receives, verifies and parses the identity token.

The results are shown in Figure 5. The overall times of SSO login are 113 ms, 310 ms, and 308 ms for MITREid Connect, UPPRESSO, and SPRESSO, respectively. In the preparation and requesting, MITREid Connect only needs 10 ms but UPPRESSO requires 271 ms. The main overhead in UPPRESSO is to open the new browser window and download the scripts, which needs about 104 ms. This overhead can be reduced by silently conducting these operations when the user visits the RP website, by the implementation with browser extensions. SPRESSO needs 19 ms in the preparation and requesting, a little more than MITREid Connect, for an RP to obtain some information on the IdP and encrypt its domain using an ephemeral symmetric key.

In the identity-token generation, UPPRESSO needs in total 34 ms. Compared with MITREid Connect, it needs 2 more ms to calculate PID_U. SPRESSO requires 71 ms to generate an identity token, as it implements the IdP based on node.js and therefore adopts a JavaScript cryptographic library, while a more efficient Java library is used in others.

In the identity-token acceptance, UPPRESSO only needs

about 6 ms for the scripts to send an identity token to the RP, which verifies it and calculates $Acct$. It takes 71 ms for MITREid Connect to accept this identity token: when the token is redirected to the RP, it must be carried within an URL, following the fragment identifier # instead of ?, due to security considerations [50], so the identity-token response has to been sent to the RP by JavaScript functions (but not HTTP requests) and most time is spent to download the script from the RP. SPRESSO needs the longest time (210 ms) due to the complicated process at the user's browser: after receiving identity tokens from the IdP, the browser downloads the JavaScript program from a trusted forwarder, decrypts the RP endpoint, and finally sends identity tokens to this endpoint.

7 Discussions

Applicability. The identity-transformation functions, i.e., $\mathcal{F}_{PID_{RP}}()$, $\mathcal{F}_{PID_U}()$, and $\mathcal{F}_{Acct}()$, are applicable to various SSO scenarios (e.g., web application, mobile App, and even native software), because these designs do not depend on any special implementation or runtime environment. Although the prototype system runs for web applications, it is feasible to apply the identity-transformation functions to other SSO scenarios to protect user privacy.

Compatibility with the Authorization Code Flow. In the authorization code flow of OIDC [1], the IdP does not directly issue the identity token; instead, an access token is forwarded to the RP, and then the RP uses this access token to ask for identity tokens from the IdP. The identity-transformation functions $\mathcal{F}_{PID_U}$, $\mathcal{F}_{PID_{RP}}$ and $\mathcal{F}_{Account}$ can be integrated into the authorization code flow to generate and verify identity tokens, so that the privacy threats are still prevented in the generation and verification of identity tokens. Accordingly, only PID_{RP} but not ID_{RP} is enclosed in the access tokens (as well as the identity tokens). However, as the RP receives the identity token directly from the IdP in this flow, it allows the IdP to obtain the RP's network information (e.g., IP address). Therefore, to prevent this leakage in the authorization code flow, UPPRESSO needs to integrate anonymous networks (e.g., Tor) for the RP to ask for identity tokens.

Implementation with Browser Extensions. To improve the portability of user agents, the user functions of UPPRESSO are implemented by browser scripts in the prototype system. However, these functions can be implemented with browser extensions, which will result in much better performance. In this case, a user downloads and installs the browser extension, before he visits an RP. After some experiments while the IdP and RPs in the prototype system are unmodified, we find that at least 102 ms will be saved for each instance (i.e., about 208 ms in total for a login instance), compared with the version implemented with browser scripts.

Collusive Attacks by the IdP and RPs. If the IdP is kept curious-but-honest and shares messages in the login flow (i.e., ID_U, PID_{RP}, $PEnpt_U$, $H(t)$, and PID_U) with some collusive RPs, UPPRESSO still provides secure SSO services, provided that the signed identity tokens are sent to the authenticated users only. Moreover, in this case, a user's login activities at the other honest RPs, are still protected from the IdP and these collusive RPs, because a triad of t, PID_U and PID_{RP} is ephemeral and independent of each other.

8 Conclusion

In this paper, we propose UPPRESSO, an untraceable and unlinkable privacy-preserving single sign-on system, to protect a web user's login activities at different RPs against both the curious IdP and collusive RPs. To the best of our knowledge, UPPRESSO is the first practical approach that defends against both the privacy threats of IdP-based login tracing and RP-based identity linkage. To achieve these goals, we convert the identity dilemma in privacy-preserving SSO services into an identity-transformation challenge and design three functions satisfying the requirements, where $\mathcal{F}_{PID_{RP}}$ protects the RP's identity from the curious IdP, $\mathcal{F}_{PID_U}$ prevents collusive RPs from linking a user based on his identities at these RPs, and $\mathcal{F}_{Acct}$ allows the RP to derive an identical account for a user in his multiple login instances. The three functions can be integrated with existing SSO protocols, such as OIDC, to enhance the protections of user privacy, while without breaking any security guarantees of existing SSO systems. The experimental evaluation of the UPPRESSO prototype demonstrates that it provides efficient SSO services, where a login instance takes only 310 ms on average.

References

[1] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, *OpenID Connect Core 1.0 incorporating errata set 1*, The OpenID Foundation, 2014.

[2] D. Hardt, *RFC 6749: The OAuth 2.0 authorization framework*, Internet Engineering Task Force, 2012.

[3] J. Hughes, S. Cantor, J. Hodges, F. Hirsch, P. Mishra, R. Philpott, and E. Maler, *Profiles for the OASIS security assertion markup language (SAML) V2.0*, OASIS, 2005.

[4] E. Maler and D. Reed, "The venn of identity: Options and issues in federated identity management," *IEEE Security & Privacy*, vol. 6, no. 2, pp. 16–23, 2008.

[5] P. Grassi, E. Nadeau, J. Richer, S. Squire, J. Fenton, N. Lefkovitz, J. Danker, Y.-Y. Choong, K. Greene, and M. Theofanos, *SP 800-63C: Digital identity guidelines: Federation and assertions*, National Institute of Standards and Technology, 2017.

[6] D. Fett, R. Küsters, and G. Schmitz, "Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the Web," in *20th European Symposium on Research in Computer Security (ESORICS)*, 2015, pp. 43–65.

[7] D. Fett, R. Küsters, and G. Schmitz, "SPRESSO: A secure, privacy-respecting single sign-on system for the Web," in *22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 1358–1369.

[8] "Google Identity Platform," https://developers.google.com/identity/, Accessed August 20, 2019.

[9] "About Firefox Accounts," https://mozilla.github.io/application-services/docs/accounts/welcome.html, Accessed August 20, 2019.

[10] N. Sakimura, J. Bradley, and M. Jones, *OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1*, The OpenID Foundation, 2014.

[11] T. Hardjono and S. Cantor, *SAML V2.0 subject identifier attributes profile version 1.0*, OASIS, 2018.

[12] Mozilla Developer Network (MDN), "Persona," https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona.

[13] Zhiyi Zhang, Michal Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière, "EL PASSO: Efficient and lightweight privacy-preserving single sign on," *Privacy Enhancing Technologies*, vol. 2021, no. 2, pp. 70–87, 2021.

[14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha, "Algebraic MACs and keyed-verification anonymous credentials," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[15] Marios Isaakidis, Harry Halpin, and George Danezis, "UnlimitID: Privacy-preserving federated identity management using algebraic MACs," in *2016 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2016*, 2016, pp. 139–142.

[16] Arkajit Dey and Stephen Weis, "PseudoID: Enhancing privacy for federated login," in *10th Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[17] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of OAuth 2.0," in *2016 ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 1204–1215.

[18] D. Fett, R. Küsters, and G. Schmitz, "The web sso standard openid connect: In-depth formal security analysis and security guidelines," in *30th IEEE Computer Security Foundations Symposium (CSF)*, 2017, pp. 189–202.

[19] D. Fett, R. Küsters, and G. Schmitz, "An expressive model for the web infrastructure: Definition and application to the BrowserID SSO system," in *35th IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 673–688.

[20] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for Google Apps," in *6th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, 2008, pp. 1–10.

[21] 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 *33rd IEEE Symposium on Security and Privacy (S&P)*, 2012, pp. 365–379.

[22] San-Tsai Sun and Konstantin Beznosov, "The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems," in *19th ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 378–390.

[23] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti, "An authentication flaw in browser-based single sign-on protocols: Impact and remediations," *Computers & Security*, vol. 33, pp. 41–58, 2013.

[24] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

[25] W. Li and C. Mitchell, "Analysing the security of Google's implementation of OpenID Connect," in *13th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016, pp. 357–376.

[26] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking SAML: Be whoever you want to be," in *21th USENIX Security Symposium*, 2012, pp. 397–412.

[27] H. Wang, Y. Zhang, J. Li, and D. Gu, "The achilles heel of OAuth: A multi-platform study of OAuth-based authentication," in *32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016, pp. 167–176.

[28] C. Mainka, V. Mladenov, and J. Schwenk, "Do not trust me: Using malicious IdPs for analyzing and attacking single sign-on," in *1st IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 321–336.

[29] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich, "Sok: Single sign-on security - An evaluation of OpenID Connect," in *2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 251–266.

[30] R. Yang, W. C. Lau, J. Chen, and K. Zhang, "Vetting single sign-on SDK implementations via symbolic reasoning," in *27th USENIX Security Symposium*, 2018, pp. 1459–1474.

[31] Y. Zhou and D. Evans, "SSOScan: Automated testing of web applications for single sign-on vulnerabilities," in *23rd USENIX Security Symposium*, 2014, pp. 495–510.

[32] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu, "Model-based security testing: An empirical study on OAuth 2.0 implementations," in *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016, pp. 651–662.

[33] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization," in *22th USENIX Security Symposium*, 2013, pp. 399–314.

[34] J. Navas and M. Beltrán, "Understanding and mitigating OpenID Connect threats," *Computer & Security*, vol. 84, pp. 1–16, 2019.

[35] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "OAuth demystified for mobile application developers," in *2014 ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 892–903.

[36] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of OAuth implementations in Android applications," in *31st Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 61–70.

[37] R. Yang, W. C. Lau, and S. Shi, "Breaking and fixing mobile app authentication with OAuth2.0-based protocols," in *15th International Conference on Applied Cryptography and Network Security (ACNS)*, 2017, pp. 313–335.

[38] S. Shi, X. Wang, and W. C. Lau, "MoSSOT: An automated blackbox tester for single sign-on vulnerabilities in mobile applications," in *14th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2019, pp. 269–282.

[39] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "O single sign-off, where art thou? An empirical analysis of single sign-on account hijacking and session management on the web," in *27th USENIX Security Symposium*, 2018, pp. 1475–1492.

[40] K. Elmufti, D. Weerasinghe, M. Rajarajan, and V. Rakocevic, "Anonymous authentication for mobile single sign-on to protect user privacy," *International Journal of Mobile Communications*, vol. 6, no. 6, pp. 760–769, 2008.

[41] Jingquan Wang, Guilin Wang, and Willy Susilo, "Anonymous single sign-on schemes transformed from group signatures," in *5th International Conference on Intelligent Networking and Collaborative Systems*, 2013, pp. 560–567.

[42] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer, "Anonymous single-sign-on for n designated services with traceability," in *23rd European Symposium on Research in Computer Security (ESORICS)*, 2018, pp. 470–490.

[43] Tian-Fu Lee, "Provably secure anonymous single-sign-on authentication mechanisms using extended chebyshev chaotic maps for distributed computer networks," *IEEE Systems Journal*, vol. 12, no. 2, pp. 1499–1505, 2018.

[44] Jinguang Han, Liqun Chen, Steve A. Schneider, Helen Treharne, Stephan Wesemeyer, and Nick Wilson, "Anonymous single sign-on with proxy re-verification," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 223–236, 2020.

[45] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Krügel, Giovanni Vigna, and Yan Chen, "Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel," in *17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014, pp. 276–298.

[46] M. Shehab and F. Mohsen, "Towards enhancing the security of OAuth implementations in smart phones," in *3rd IEEE International Conference on Mobile Services*, 2014, pp. 39–46.

[47] Shafi Goldwasser and Yael Tauman Kalai, "Cryptographic assumptions: A position paper," in *Theory of Cryptography - 13th International Conference, TCC 2016-A*, 2016, vol. 9562, pp. 505–522.

[48] "MITREid connect /openid-connect-java-spring-server," https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server, Accessed August 20, 2019.

[49] "jsrsasign," https://kjur.github.io/jsrsasign/, Accessed August 20, 2019.

[50] B de Medeiros, Marius Scurtescu, Paul Tarjan, and M Jones, "Oauth 2.0 multiple response type encoding practices," *OpenID Specification*, 2014.

A The Security Proofs of UPPRESSO

We formally analyze the security properties of UPPRESSO based on the Dolev-Yao style web model as follows. We first describe the expressive Dolev-Yao style model of the web infrastructure. Then, we formulate the UPPRESSO system using this model, including all entities (i.e., browsers, servers, and scripts) and the transmitted data. Finally, we finish the security proofs of UPPRESSO by tracing the identity tokens in this model.

A.1 The Dolev-Yao Style Web Model

Based on Dolev-Yao style models, the web infrastructure is modelled as the form of $(\mathcal{W}, \mathcal{S}, E^0)$. These notations are explained as below.

- $\mathcal{W}$ is the set of *atomic processes*. An atomic process represents an independent entity in the web system, such as browser and web server.
- $\mathcal{S}$ is the set of *script processes*. Besides the atomic processes in the web system, there are also entities processing data and communicating with other entities, but not running independently without atomic processes. These entities are called script processes, e.g., JavaScript code within a browser.
- E^0 is the set of self-triggering *events* acceptable to the processes in $\mathcal{W}$. Events are the basic communication elements in the model, representing a message sent by a process to another process. The self-triggering event represents the start of the procedure in the web system. For example, while the script is loaded by the browser, the browser would generate the self-triggering event to start the script.

Term. Terms are the basic elements in Dolev-Yao style models. It may contain constants such as ASCII strings and integers, sequence symbols such as k-ary sequences ($k \leq 0$), or function symbols that model cryptographic primitives such as `SigSign`, `SigVerify` and `Hash`. For example, an HTTP request is expressed as a term containing a type (e.g., `HTTPReq`), a nonce, a method (e.g., `GET` or `POST`), a domain, a path, URL parameters, request headers and a message body. So, an HTTP GET request exa.com/path?para=1 with empty headers and bodys is expressed as $\langle \texttt{HTTPReq}, n, \texttt{GET}, exa.com, /path, \langle\langle para, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$.

Operations over terms. The operations are defined.

- **Equational theory**. Equational theory uses the symbol $\equiv$ to represent the congruence relation on terms, and $\not\equiv$ for non-congruence relation. For example, while there are the data *Data*, its signature *Sig*, and the corresponding public key *PK*, the relation is described as $\texttt{SigVerify}(Data, Sig, PK) \equiv \texttt{TRUE}$.
- **Patten Matching**. We define the term with the variable $*$ as the pattern, such as $\langle a, b, * \rangle$. The pattern matches any term which only replaces the $*$ with other terms. For instance, $\langle a, b, * \rangle$ matches $\langle a, b, c \rangle$.
- **Retrieve attributes from formatted term**. Formatted term is the data in the specific format, for instance, the HTTP request is the formatted data in the form $\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body \rangle$. We assume there is an HTTP request $r := \langle \texttt{HTTPReq}, n, \texttt{GET}, example.com, /path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$, here we define the operation on the r. That is, the elements in r can be accessed in the form *r.name*, such that $r.method \equiv \texttt{GET}$, $r.path \equiv /path$ and $r.body \equiv \langle\rangle$.
- **Retrieve attributes from dictionary term**. Dictionary term is the data in the form $\langle\langle type, value\rangle, \langle type, value\rangle, \ldots\rangle$, for instance the *body* in HTTP request is dictionary data. We assume there is a $body := \langle\langle username, alice\rangle, \langle password, 123\rangle\rangle$, here we define the operation on the *body*. That is, we can access the elements in *body* in the form $body[name]$, such that $body[username] \equiv alice$ and $body[password] \equiv 123$. We can also add the new attributes to the dictionary, for example after we set $body[age] := 18$, the *body* are changed into $\langle\langle username, alice\rangle, \langle password, 123\rangle, \langle age, 18\rangle\rangle$.

State. State is the term consisted of basic terms, to sketch the atomic process at the point of time. For example, the state of a login server is described as the term $\langle SessionList, UserList \rangle$, while the *SessionList* maintains the visitors' cookies with the corresponding login state, and the $UserList$ maintains all user identities (either logined or unlogined) and their corresponding credential verifiers.

Relation. The relation represents the model of procedure, showing how the entity deal with the received message. For example, while the login server receive the login request from a visitor (the request is denoted as the event e), the input of the relation is e and current state $\langle SessionList, UserList \rangle$. The server verifies the visitor's identity with the $UserList$ and add the login result into *SessionList* (transitioning to $SessionList'$). Thus the output is the login response e' and the new state $\langle SessionList', UserList \rangle$.

Event. An event is expressed formally as a term $\langle a, b, m \rangle$, where a and b represent the addresses of the sender and receiver, respectively, and m is the message transmitted.

Atomic process. An atomic process is expressed as (I^p, Z^p, R^p, z_0^p). The elements are explained as below.

- I^p is the set of *addresses* that the process listens to.
- Z^p is the set of *states* that describes the process.

- R^p is the relation that models the computations of the process, which receives an event in a state and then moves to another state and outputs a set of events.
- z_0^p is the initial state.

Script process. A script process is the formulation of browser scripts as a relation R, representing the server-defined functions, while the input and output are handled by the browser. The script process does not maintain any state itself, while the `scriptstate` is stored by the browser in each documents. The `scriptstate` is transmitted to script process as the part of input, when the script process is invoked to process the event. A script process is addressed by the `Origin` property, as the the protocol and hostname of a URL. That is, while a script sends the messages to another script process and set the `Origin` value as the IdP domain, only the script downloaded from IdP server receives this message.

A.2 The Formulation of UPPRESSO

In this section, we introduce the model of UPPRESSO, in the form of $(\mathcal{W}, \mathcal{S}, E^0)$. $\mathcal{W}$ is the set of atomic processes, including IdP processes, RP processes and browsers. $\mathcal{S}$ is the set of script processes, containing of IdP script process and RP script process. E^0 is the set of self-triggering events acceptable to the processes in $\mathcal{W}$. In the UPPRESSO system, the self-triggering events are sent while the RP and IdP scripts are loaded. The self-triggering event for RP script is used to create the new window for download IdP script. The self-triggering event for IdP script is used to start the PID_{RP} negotiation.

We only focus on the model of servers, browsers and the plain message transmissions among them in this paper, and neglect the HTTPS requests and other necessary Internet facilities (e.g., DNS server) deployed in the UPPRESSO system. Because we assume that the internet architecture is well built and HTTPS is well implemented, so that an adversary cannot conduct the attacks targeting these layers.

A.2.1 The Data Transmitted and Processed

We provide the modelling normal data, such HTTP messages, used to construct UPPRESSO model.

HTTP Messages. An HTTP request message is the term of the form

$$\langle \texttt{HTTPReq}, nonce, method, host, path, \\ parameters, headers, body \rangle$$

An HTTP response message is the term of the form

$$\langle \texttt{HTTPResp}, nonce, status, headers, body \rangle$$

The details are defined as follows:

- `HTTPReq` and `HTTPResp` denote the types of messages.
- *nonce* is a random number that maps the response to the corresponding request.
- *method* is one of the HTTP methods, `GET` and `POST`.
- *host* is the constant string domain of visited server.
- *path* is the constant string representing the concrete resource of the server.
- *parameters* contains the parameters carried by the url as the form $\langle\langle type, value\rangle, \langle type, value\rangle, \ldots\rangle$, for example, the *parameters* in the url $http://www.example.com?type=confirm$ is $\langle\langle type, confirm\rangle\rangle$.
- *headers* is the header content of each HTTP messages as the form $\langle\langle type, value\rangle, \langle type, value\rangle, \ldots\rangle$, such as $\langle\langle Referer, http://www.example.com\rangle, \langle Cookies, c\rangle\rangle$.
- *body* is the body content carried by HTTP `POST` request or HTTP response in the form $\langle\langle type, value\rangle, \langle type, value\rangle, \ldots\rangle$.
- *status* is the HTTP status code defined by HTTP standard, such as 200, 302 and 404.

URL. URL is a term $\langle \texttt{URL}, protocol, host, path, parameters\rangle$, where `URL` is the type, *protocol* is chosen in {`S`, `P`} as `S` stands for HTTPS and `P` stands for HTTP. The *host*, *path*, and *parameters* are the same as in HTTP messages.

Origin. An Origin is a term $\langle host, protocol\rangle$ that can represent the server that a script is downloaded from, where *host* and *protocol* are the same as in URL.

POSTMESSAGE. PostMessage is used in the browser for transmitting messages between scripts. The postMessage package is defined as the form $\langle \texttt{POSTMESSAGE}, target, Content, Origin\rangle$, where `POSTMESSAGE` is the type, *target* is the constant nonce which stands for the receiver, *Content* is the message transmitted and *Origin* restricts the receiver.

XMLHTTPREQUEST. XMLHTTPRequest is the HTTP message transmitted by scripts in the browser. That is, the XMLHTTPRequest is converted from the HTTP message by the browser. The XMLHTTPRequest package is defined as the term in the form $\langle \texttt{XMLHTTPREQUEST}, URL, methods, Body, nonce\rangle$ can be converted into HTTP request message by the browser, and $\langle \texttt{XMLHTTPREQUEST}, Body, nonce\rangle$ is converted from HTTP response message.

A.2.2 The Atomic Processes of IdP and RP

An atomic process is a tuple $p = (I^p, Z^p, R^p, z_0^p)$, containing the addresses, states, relation and the initial state.

IdP Server Process. The state of IdP process is described as a term in the form $\langle Issuer, SK, SessionList, UserList, RPList, Validity, TokenList\rangle$. That is, the state of an IdP server at the point in time can be sketched using these attributes.

- *Issuer* is the identifier of IdP.
- *SK* is the private key used by IdP to generate signatures.
- *SessionList* is the term in the form of $\langle\langle cookie, session\rangle\rangle$, the cookie uniquely identifies the session storing the browser uploaded messages.
- *UserList* is the set of user's information, including ID_U and other attributes.
- *RPList* is the set of registered RP information which consists of ID of RP (PID_{RP}), *Enpt* and *Validity*.
- *Validity* is the validity for IdP generated signatures.
- *TokenList* is the set of IdP generated Identity tokens.

We also define the *functions* in the relations.

- `CredentialVerify(credential)` is used to authenticate the user.
- `UIDOfUser(credential)` is used to search the user's ID_U.
- `ListOfPID()` is the set of PIDs of registered RP in some login instance.
- `EndpointsOfRP(r)` is the endpoint registered by the IdP script with PID *r*.
- `Multiply(P,a)` is the result of aP, where P is the point on elliptic curve and a is the integer.
- `CurrentTime()` is the system current time.

The relation of IdP process R^i is shown as Relation 1 in Appendix B.

RP server process. The state of RP process is the formatted term $\langle ID_{RP}, Enpt, IdP, Cert, SessionList, UserList\rangle$.

- ID_{RP} and *Enpt* are RP's registered information at IdP.
- *Cert* is the IdP-signed message containing ID_{RP}, *Enpt* and other attributes.
- *IdP* is the term of the form $\langle ScriptUrl, PK\rangle$, where *ScriptUrl* is the site to download IdP script, and *PK* is the public key used to verify the IdP signed messages.
- *SessionList* is same as it in IdP process.
- *UserList* is the set of users registered at this RP, each user is uniquely identified by the *Acct*.

The *functions* are defined as follows:

- `Inverse(t)` calculates the trapdoor in UPPRESSO system.
- `Random()` generates a fresh random number.
- `AddUser(Acct)` add the new user with *Acct* into RP's user list.

The relation of RP process R^r is shown as Relation 2 in Appendix B.

A.2.3 Browser

We analyze how the browsers interact with other parties in UPPRESSO. In UPPRESSO the message transmitted through a browser is constructing and parsing by the script, so that we only focus on how the script process runs in the browser.

In the browser, a window is the basic unit that shows the content to user. In a window, the document represents the whole page contained in this window. The script is one node in the document. We introduce the windows and documents of the browser model, which provides inputs and parses the outputs of the script process. The opened windows, documents and downloaded scripts are parts of the state of a browser.

Window. A window w is a formatted term $w = \langle ID_w, documents, opener\rangle$, representing the concrete browser window in the system.

- The ID_w is the window reference to identify each windows.
- The *documents* is the set of documents (defined below) including the current document and cached documents (for example, the documents can be viewed via the "forward" and "back" buttons in the browser).
- The *opener* represents the document from which this window is created, for instance, while a user clicks the href in document d and it creates a new window w, there is $w.opener \equiv d.ID_d$.

Document. A document d is the web page content in the browser window, in the formatted term

$$\langle ID_d, location, referrer, script, scriptstate, scriptinputs, subwindows\rangle$$

- The ID_d locates the document.
- *Location* is the URL where the document is loaded.
- *Referrer* is same as the Referer header defined in HTTP standard.
- The *script* is the script process downloaded from each servers.
- *scriptstate* is define by the script, different in each scripts.
- The *scriptinputs* is the message transmitted into the script process.
- The *subwindows* is the set of ID_w of document's created windows.

A.2.4 The IdP Script and the RP Script

The script process is the dependent process relying on the browser, which can be considered as a relation R mapping a message input and a message output. And finally the browser will conduct the command in the output message. A script can be modelled as the tuple containing the input, output and relation.

IdP script process. At the first, we give the description of the formate of input. The input is the formatted term

$$\langle tree, docID, scriptstate, stateinputs, cookies, ids, secret \rangle$$

- The *tree* is the structure of the opened windows and documents, which are visible to this script.
- *DocID* is the document ID_d, representing a document. *Scriptinputs* is the message transmitted to script.
- The *scriptinputs* is defined as formatted term, for example, postMessage is one of the forms of *scriptinputs*.
- *Cookies* is the set of cookies that belong to the document's origin.
- *Ids* is the set of users while *secret* is the credential of corresponding user.

The state of IdP script process *scriptstate* is the formatted term

$$\langle IdPDomain, Parameters, phase, refXHR \rangle$$

- *IdPDomain* is the IdP's host.
- *Parameters* is used to store the parameters received from other processes.
- *phase* is used to label the procedure point in the login.
- *refXHR* is the nonce to map HTTP request and response.

The output is the formatted term

$$\langle scriptstate, cookies, command \rangle$$

The *scriptstate* and *cookies* are transitioned to a new state. The *command* is the operation which is to be conducted by the browser. Here we only introduce the form of commands used in UPPRESSO. We have defined the postMessage and XMLHTTPRequest (for HTTP request) message which are the *commands*. Moreover, a formatted term $\langle \texttt{IFRAME}, URL, WindowID \rangle$ asks the browser to create this document's subwindow and it visits the server with the URL.

The relation of IdP script process *script_idp* is shown in Appendix B Relation 3. The new *functions* in relation are defined as follows.

- `PARENTWINDOW`$(tree, docID)$. The first parameter is the input relation tree defined before, and the second parameter is the ID_d of a document. The output returned by the function is the current window's opener's ID_w (null if it doesn't exist nor it is invisible to this document).
- `CHOOSEINPUT`$(inputs, pattern)$. The first parameter is a set of messages, and the second parameter is a pattern. The result returned by the function is the message in *inputs* matching the *pattern*.
- `RandomUrl`$()$ returns a newly generated host string.
- `CredentialofUser`$(user, secret)$ creates the user's credential for authentication.

RP script process The input is the formatted term

$$\langle tree, docID, scriptstate, stateinputs, cookies \rangle$$

While the output is the formatted term

$$\langle scriptstate, cookies, command \rangle$$

The state of RP script process *scriptstate* is the formatted term $\langle IdPDomain, RPDomain, Parameters, phase, refXHR \rangle$. The *RPDomain* is the host string of the corresponding RP server, and other terms are defined in the same way as in IdP script process. Other terms are same as them in IdP script

The relation of RP script process *script_rp* is shown in Appendix B Relation 4. We define the function `SUBWINDOW`$(tree, docID)$, which takes the *tree* defined above and the current document's ID_d as the input. And it selects the ID_w of the first window opened by this document as the output. However, if there is no opened windows, it returns null.

A.3 Proofs of Security

A secure SSO service ensures that only the legitimate user logins to an honest RP under his unique account.

Definition 1. An SSO system is secure, if and only if, in an honest user's all login instances to an honest RP, the derived accounts are identical and unique to any accounts derived in other login instances to this RP (by either honest or malicious entities).

Theorem 1. *When the IdP is honest, the following properties of an identity token which is verified by an RP to derive the account in a login instance, result in a secure SSO system:*

- **Integrity** *An honest RP accepts only identity tokens issued by the IdP, and any forged or modified identity token will be rejected by the RP.*
- **Confidentiality** *An identity token issued by the Idp, is accessible to only the authenticated user and the target RP, in addition to the IdP.*

- **User Identification** *The user (pseudo-)identity bound in the identity token identifies the user authenticated by the IdP, and only this user, within its validity.*

- **RP Designation** *The RP (pseudo-)identity bound in the identity token identifies the target RP, and only this RP, within its validity.*

Proof. These four properties are sufficient conditions of a secure SSO system. First of all, integrity ensures that any identity token acceptable to the honest RP, is issued the IdP. Then, provided that the IdP is honest, confidentiality further ensures that only the authenticated user and the target RP of an acceptable identity token are able to present this identity token in some login instances, collusive with adversaries sometimes.

User identification enables the honest RP to identify the authenticated user in the verification of an identity token and an honest user only presents identity token accessible to himself, so that the derived accounts in the honest user's all login instances are identical and unique to the accounts derived in other honest users' login instances; otherwise, a token does not identify the authenticated user and user identification is violated.

Because an honest RP accepts only identity tokens designating itself, RP designation prevents an identity token from being accept by an honest RP which is not the target. Thus, although confidentiality does not prevent a malicious user or RP from presenting the identity token in attacks, (*a*) such an identity token will not be accepted by any honest RP, if it is presented by the malicious RP which is the target RP of this token, and (*b*) if such an identity token is presented by the malicious user and accepted by an honest RP which is designated in this token, only this malicious user's account will be derived and then unique to others, for user identification ensures only the authenticated user is identified. Therefore, in an honest user's some login instance to an honest RP, the derived account is unique to any accounts derived in even a malicious user's login instances to this RP.

Finally, as an honest RP accepts only unexpired identity token, the within-its-validity identification requirements of user identification and RP designation are sufficient. □

We assume that in UPPRESSO system, all the network messages are transmitted using HTTPS, postMessage messages are protected by the browser, and the browsers are honest, so web attackers can never break the security of UPPRESSO. Here, we are going to prove that, in the UPRESSO system, the following lemmas are always workable.

Lemma 1. In the UPPRESSO system, an identity token issued by the Idp, is accessible to only the authenticated user and the target RP, in addition to the IdP.

Proof. Here we only need to prove that attackers cannot retrieve the token from other non-collusive processes.

- Attacker cannot obtain the identity token from RP server. We check all the messages sent by the RP server at Line 4, 7, 19, 25, 31, 36, 45, 55, 61, 66, 74, 84 in Relation 2. It is easy to prove that the RP server does not send any token to other processes.
- Attacker cannot obtain the identity token from RP script. The messages sent by RP script can be classified into two classes. 1) The messages at Line 18, 36, 56 in Relation 4 are sent to the RPDomain which is set at Line 4, so that attackers cannot receive these messages. 2) The messages at Line 26, 46 only carry the contents received from RP server, and we have proved that RP server does not send any token. Therefore, attackers cannot receive the *Token* from RP script.
- Attacker cannot obtain the identity token from IdP server. Considering the messages at Line 4, 11, 15, 22, 25, 35, 43, 50, 66 in Relation 1, we find that only the message at Line 66 carries the token. This token is generated at Line 64, following the trace of the *Content* at Line 62, the PID_U at Line 60, the ID_U at Line 59, the *session* at Line 47, and finally the *cookie* at Line 46. That is, the receiver of token must be the owner of the *cookie* in which session that saves the parameter ID_U. The ID_U is set at Line 14 after verifying the credential and never modified. As we assume that credential cannot be known to attackers, attackers cannot obtain the identity token from IdP server.
- Attacker cannot obtain the identity token from IdP script. As the proof provided above, only IdP sends the token with the message at Line 66 in Relation 1, the IdP script can only receive the token at Line 99 in Relation 3. Here we are going to prove that the token issued for user u's login to RP r (denoted as $t(u,r)$) can only be sent to the corresponding RP server through IdP script. The receiver of $t(u,r)$ is restricted by the *RPOrigin* at Line 100, which is set at Line 55. The host in the *RPOrigin* is verified using the one included in *Cert* at Line 51. If the *Cert* belong to r, the attacker cannot obtain the $t(u,r)$. Now we give the proof that the *Cert* belongs to r. Firstly we define the negotiated PID_{RP} in $t(u,r)$ as p. That is the PID_{RP} at Line 69 in Relation 2 must equal to p and the PID_{RP} is verified at Line 44 with the *RegistrationToken*. This verification cannot be bypassed due to the state check at Line 60. At the same validity period, the IdP script needs to send the registration request with same p and receive the successful PID_{RP}-registration result. As the IdP checks the uniqueness of PID_{RP} at Line 31 in Relation 1. The r and IdP script must share the same *RegistrationToken*. As the *RegistrationToken* contains the $\texttt{Hash}(t)$, the IdP script and r must share the same ID_{RP}. Therefore, the *Cert* saved as the IdP scriptstate parameter must belong to r. Thus, the identity token would only be sent to the script in the corresponding RP's origin.

Therefore, attackers cannot learn users' valid identity tokens. This lemma is proved. □

Lemma 2. In the UPPRESSO system, the RP (pseudo-)identity bound in the identity token identifies the target RP, and only this RP, within its validity.

Proof. We can find that an RP server finally accept the PID_U in an token at Line 77 in Relation 2, after the verification at Line 73. While PID_U is accepted, the PID_{RP} must equal with the stored one retrieved at Line 69 and verified at Line 44, Relation 2. The stored PID_{RP} is included in the registration result verified at Line 43, Relation 2, received from IdP server. Thus, the PID_{RP} is only valid at the RP received the corresponding registration result.

Moreover, this registration result is generated at Line 40, Relation 1 and protected by the signature generated at Line 41. According to the verification at 31, Relation 1, the PID_{RP} is unique during the valid time, so that no other registration results containing this PID_{RP} exists. Moreover, according to Line 42, 44 (the t used at Line 42 is set at Line 15, used for generating PID_{RP} at Line 13), the PID_{RP} must be resigned for this RP, as no other RP cannot generate the same PID_{RP} with this t. So only one RP is able to achieve the corresponding registration result of an PID_{RP}.

Based on the two facts, (1) the PID_{RP} is only valid at the RP received the corresponding registration result, and (2) only one RP is able to achieve the corresponding registration result of an PID_{RP}. It can be inferred that the PID_{RP} is only valid at one specific RP.

Therefore, this lemma is proved.

□

Lemma 3. In the UPPRESSO system, an honest RP accepts only identity tokens issued by the IdP, and any forged or modified identity token will be rejected by the RP.

Proof. The *Acct* derived from the identity token by an RP, is generated based on the t^{-1} according to Line 79, Relation 2. It can be found out that, the integrity of t^{-1} is protected by the IdP issued registration result, according to Line 41-44, Relation 2. The integrity of ID_{RP} finally contained in the derived *Acct* is protected by *Cert*, according to Line 15-21, Relation 3. Thus, here we would prove the integrity of identity token, RP *Cert* and registration result. We can easily find that the IdP does not send the private key to any processes so that the attackers cannot obtain the private key. Then we only need to prove that all the proofs are well verified.

- *Cert* is used at Line 21, 52 in Relation 3. At Line 21, the *Cert* has already been verified at Line 16. At Line 52, the *Cert* is picked from the state parameters, and the cert parameter is set at Line 19. At Line 19, the *Cert* has already been verified at Line 16. At Line 16 the *Cert* is verified with the public key in the scriptstate, where the key is considered initially honest and the key is not modified at Relation 3. Therefore, *Cert* cannot be forged or modified.
- *RegistrationResult* is used in Relation 2 from Line 35 to 55, which is verified at Line 30. The public key is initially set in the RP and never modified. Therefore, *RegistrationResult* cannot be forged or modified.
- *Token* is used in Relation 2 from Line 69 to 84 after Line 65 where it is verified. As proved before, the public key is honestly set and never modified. Therefore, *Token* cannot be forged or modified.

Therefore, this lemma is proved. □

Lemma 4. In the UPPRESSO system, the user (pseudo-)identity bound in the identity token identifies the user authenticated by the IdP, and only this user, within its validity.

Proof. We can find that, the RP accepts the user's identity at Line 83, Relation 2. And the identity is generated at Line 79, based on the PID_U retrieved from the token and the trapdoor t^{-1}. The t^{-1} is generated at Line 14, set at Line 17, and never changed, as the multiplicative inverse of t. According to Lemma 3, only the IdP can generate an token, so that the token must be issued at Line 64, Relation 1. IdP generates the PID_U based on the PID_{RP} and user's ID_U. According to Lemma 2, while the PID_{RP} is accepted by an RP, the PID_{RP} must be the generated between the RP and user. According to Lemma 1, the ID_U must belongs to the honest user. Therefore, the user's identity must equal with the constant $[ID_U]ID_{RP}$, and unique to other users' *Acct*s at this RP.

Moreover, an adversary may allure the honest user to upload the adversary's token to an RP, so that the honest user may use the same *Acct* as the adversary's. While the honest user has already negotiated the PID_{RP} with the RP, the opener of the IdP script must be the RP script. As the t generated at Line 7, Relation 3, and PID_{RP} generated at Line 21, Relation 3 and Line 13, Relation 2. The t is only sent to RP script at Line 8, Relation 3, and RP server at Line 18, Relation 4. The PID_{RP} is only sent to IdP server at Line 28 and 90, Relation 3. Moreover, the PID_{RP}-registration result including the PID_{RP} is sent to IdP script at Line 35, Relation 1, RP script at Line 40, Relation 3, RP server at Line 36, Relation 4. An adversary can never know the PID_{RP} negotiated between the honest user and RP. While the honest user has not negotiated the PID_{RP} with the RP yet, the `session`$[PID_{RP}]$ at Line 69, Relation 2 and other similar attributes must be empty, as the *Cookie* used at Line 58 belongs to the honest user and the negotiation between Line 10-20 does not conducted by the user. Therefore, an adversary cannot lead the honest RP to accept the malicious token from the honest user.

This lemma is proved. □

Theorem 2. *UPPRESSO is a secure SSO system, if the IdP is honest.*

Proof. In conclusion, four lemmas prove that the UPPRESSO system satisfies the sufficient requirements of the secure SSO system. Therefore, UPPRESSO is proved secure. □

B Relations

Relation takes an event and a state as input and returns a new state and a sequence of events. In a web system, it represents how the entities, such as the server, deal with the received messages, and send message to other entities. This section provides the detailed relations of the processes (including atomic and script processes) in UPPRESSO.

B.1 IdP process

The IdP process only accepts the events, downloading scripts, accessing the login status of a cookie owner, authenticating with password, registering the PID_{RP}, and requiring the identity token. The detailed procedure of dealing with these events is shown as follows.

1 IdP_Process_Relation

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

let $s' := s$
let $n, method, path, parameters, headers, body$ **such that**
  $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$
  if *possible*; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{IdPScript}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /login$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $credential := body[credential]$
  if $CredentialVerify(credential)$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure}\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $session[uid] := \texttt{UIDOfUser}(credential)$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /loginInfo$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $uid := session[uid]$
  if $uid \not\equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Logged}\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unlogged}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /dynamicRegistration$ **then**
  let $PID_{RP} := body[PID_{RP}]$
  let $Enpt := body[Enpt]$
  let $Nonce := body[Nonce]$
  if $PID_{RP} \in \texttt{ListOfPID}()$ **then**
    let $Content := \langle Fail, PID_{RP}, Nonce\rangle$
    let $Sig := \texttt{SigSign}(Content, s'.SK)$
    let $RegistrationResult := \langle Content, Sig\rangle$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $Validity := \texttt{CurrentTime}() + s'.Validity$
  let $s'.RPList := s'.RPList +^{\langle\rangle} \langle PID_{RP}, Enpt, Validity\rangle$

40: **let** $Content := \langle OK, PID_{RP}, Nonce, Validity \rangle$
41: **let** $Sig := \texttt{Sig}(Content, s'.SK)$
42: **let** $RegistrationResult := \langle Content, Sig \rangle$
43: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult \rangle$
44: **stop** $\langle b, a, m' \rangle, s'$
45: **else if** $path \equiv /authorize$ **then**
46: **let** $cookie := headers[Cookie]$
47: **let** $session := s'.SessionList[cookie]$
48: **let** $uid := session[uid]$
49: **if** $uid \equiv \texttt{null}$ **then**
50: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
51: **stop** $\langle b, a, m' \rangle, s'$
52: **end if**
53: **let** $PID_{RP} := parameters[PID_{RP}]$
54: **let** $Enpt := parameters[Enpt]$
55: **if** $PID_{RP} \notin \texttt{ListOfPID}() \vee Enpt \notin \texttt{EndpointsOfRP}(PID_{RP})$ **then**
56: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
57: **stop** $\langle b, a, m' \rangle, s'$
58: **end if**
59: **let** $ID_U := session[uid]$
60: **let** $PID_U := \texttt{Multiply}(PID_{RP}, ID_U)$
61: **let** $Validity := \texttt{CurrentTime}() + s'.Validity$
62: **let** $Content := \langle PID_{RP}, PID_U, s'.Issuer, Validity \rangle$
63: **let** $Sig := \texttt{SigSign}(Content, s'.SK)$
64: **let** $Token := \langle Content, Sig \rangle$
65: **let** $s'.TokenList := s'.TokenList +^{\langle\rangle} Token$
66: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Token, Token \rangle \rangle$
67: **stop** $\langle b, a, m' \rangle, s'$
68: **end if**
69: **stop** $\langle\rangle, s'$

B.2 RP process

The RP process only accepts the events, downloading scripts, redirecting to IdP server, negotiating the PID_{RP}, uploading the PID_{RP}-registration result and identity token. The detailed procedure of dealing with these events is shown as follows.

2 RP_Process_Relation

Input: $\langle a, b, m \rangle, s$
1: **let** $s' := s$
2: **let** $n, method, path, parameters, headers, body$ **such that**
$\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$
if *possible*; **otherwise** stop $\langle\rangle, s'$
3: **if** $path \equiv /script$ **then**
4: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript} \rangle$
5: **stop** $\langle b, a, m' \rangle, s'$
6: **else if** $path \equiv /login$ **then**
7: **let** $m' := \langle \texttt{HTTPResp}, n, 302, \langle \langle Location, s'.IdP.ScriptUrl \rangle \rangle, \langle\rangle \rangle$
8: **stop** $\langle b, a, m' \rangle, s'$
9: **else if** $path \equiv /startNegotiation$ **then**
10: **let** $cookie := headers[Cookie]$
11: **let** $session := s'.SessionList[cookie]$
12: **let** $t := parameters[t]$
13: **let** $PID_{RP} := \texttt{Multiply}(s'.ID_{RP}, t)$
14: **let** $t^{-1} := \texttt{Inverse}(t)$

15: **let** $session[t] := t$
16: **let** $session[PID_{RP}] := PID_{RP}$
17: **let** $session[t^{-1}] := t^{-1}$
18: **let** $session[state] := expectRegistration$
19: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Cert, s'.Cert \rangle \rangle$
20: **stop** $\langle b, a, m' \rangle, s'$
21: **else if** $path \equiv /registrationResult$ **then**
22: **let** $cookie := headers[Cookie]$
23: **let** $session := s'.SessionList[cookie]$
24: **if** $session[state] \not\equiv expectRegistration$ **then**
25: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
26: **stop** $\langle b, a, m' \rangle, s'$
27: **end if**
28: **let** $RegistrationResult := body[RegistrationResult]$
29: **let** $Content := RegistrationResult.Content$
30: **if** $\texttt{SigVerify}(Content, RegistrationResult.Sig, s'.IdP.PK) \equiv \texttt{FALSE}$ **then**
31: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
32: **let** $session := \texttt{null}$
33: **stop** $\langle b, a, m' \rangle, s'$
34: **end if**
35: **if** $Content.Result \not\equiv OK$ **then**
36: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
37: **let** $session := \texttt{null}$
38: **stop** $\langle b, a, m' \rangle, s'$
39: **end if**
40: **let** $PID_{RP} := session[PID_{RP}]$
41: **let** $t := session[t]$
42: **let** $Nonce := \texttt{Hash}(t)$
43: **let** $Time := \texttt{CurrentTime}()$
44: **if** $PID_{RP} \not\equiv Content.PID_{RP} \vee Nonce \not\equiv Content.Nonce \vee Time > Content.Validity$ **then**
45: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
46: **let** $session := \texttt{null}$
47: **stop** $\langle b, a, m' \rangle, s'$
48: **end if**
49: **let** $session[PIDValidity] := Content.Validity$
50: **let** $Enpt \equiv s'.Enpt$
51: **let** $session[state] := expectToken$
52: **let** $Nonce' := \texttt{Random}()$
53: **let** $session[Nonce] := Nonce'$
54: **let** $Body := \langle PID_{RP}, Enpt, Nonce' \rangle$
55: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, Body \rangle$
56: **stop** $\langle b, a, m' \rangle, s'$
57: **else if** $path \equiv /uploadToken$ **then**
58: **let** $cookie := headers[Cookie]$
59: **let** $session := s'.SessionList[cookie]$
60: **if** $session[state] \not\equiv expectToken$ **then**
61: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
62: **stop** $\langle b, a, m' \rangle, s'$
63: **end if**
64: **let** $Token := body[Token]$
65: **if** $\texttt{checksig}(Token.Content, Token.Sig, s'.IdP.PK) \equiv \texttt{FALSE}$ **then**
66: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
67: **stop** $\langle b, a, m' \rangle, s'$
68: **end if**

```
    let PID_RP := session[PID_RP]
    let Time := CurrentTime()
    let PIDValidity := session[PIDValidity]
    let Content := Token.Content
    if PID_RP ≠ Content.PID_RP ∨ Time > Content.Validity ∨ Time > PIDValidity then
      let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
      stop ⟨b, a, m'⟩, s'
    end if
    let PID_U := Content.PID_U
    let t^-1 := session[t^-1]
    let Acct := Multiply(PID_U, t^-1)
    if Acct ∉ ListOfUser() then
      let AddUser(Acct)
    end if
    let session[user] := Acct
    let m' := ⟨HTTPResp, n, 200, ⟨⟩, LoginSuccess⟩
    stop ⟨b, a, m'⟩, s'
  end if
  stop ⟨⟩, s'
```

B.3 IdP script process

The IdP script process is only invoked to process the events, (a) self-triggering events for starting PID_{RP} negotiation; (b) the postMessage from other scripts for sending the *Cert*, and request of identity token; (c) the HTTP response for transmitting PID_{RP}-registration result and identity token. The detailed procedure of dealing with these events is shown as follows.

3 IdP_Script_Relation

```
Input: ⟨tree, docID, scriptstate, scriptinputs, cookies, ids, secret⟩
  let s' := scriptstate
  let command := ⟨⟩
  let target := PARENTWINDOW(tree, docID)
  let IdPDomain := s'.IdPDomain
  switch s'.phsae do
    case start:
      let t := Random()
      let command := ⟨POSTMESSAGE, target, ⟨⟨t, t⟩⟩, null⟩
      let s'.Parameters[t] := t
      let s'.phase := expectCert
    case expectCert:
      let pattern := ⟨POSTMESSAGE, *, Content, *⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        let Cert := input.Content[Cert]
        if checksig(Cert.Content, Cert.Sig, s'.PubKey) ≡ null then
          let stop ⟨⟩
        end if
        let s'.Parameters[Cert] := Cert
        let t := s'.Parameters[t]
        let PID_RP := Multiply(Cert.Content.ID_RP, t)
        let s'.Parameters[PID_RP] := PID_RP
        let Enpt := RandomUrl()
        let s'.Parameters[Enpt] := Enpt
        let Nonce := Hash(t)
        let Url := ⟨URL, S, IdPDomain, /dynamicRegistration, ⟨⟩⟩
```

```
27:        let s'.refXHR := Random()
28:        let command := ⟨XMLHTTPREQUEST, Url, POST,
               ⟨⟨PID_RP, PID_RP⟩, ⟨Nonce, Nonce⟩, ⟨Enpt, Enpt⟩⟩, s'.refXHR⟩
29:        let s'.phase := expectRegistrationResult
30:     end if
31:  case expectRegistrationResult:
32:     let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
33:     let input := CHOOSEINPUT(scriptinputs, pattern)
34:     if input ≠ null ∧ input.Content[RegistrationResult].type ≡ OK then
35:        let RegistrationResult := input.Body[RegistrationResult]
36:        if RegistrationResult.Content.Result ≠ OK then
37:           let s'.phase := stop
38:           let stop ⟨⟩
39:        end if
40:        let command := ⟨POSTMESSAGE, target, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, null⟩
41:        let s'.phase := expectTokenRquest
42:     end if
43:  case expecTokenRquest:
44:     let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
45:     let input := CHOOSEINPUT(scriptinputs, pattern)
46:     if input ≠ null then
47:        let PID_RP := input.Content[PID_RP]
48:        let Enpt_RP := input.Content[Enpt]
49:        let s'.Parameters[Nonce] := input.Content[Nonce]
50:        let Cert := s'.Parameters[Cert]
51:        if Enpt_RP ≠ Cert.Content.Enpt ∨ PID_RP ≠ s'.Parameters[PID_RP] then
52:           let s'.phase := stop
53:           let stop ⟨⟩
54:        end if
55:        let s'.Parameters[Enpt_RP] := Enpt_RP
56:        let Url := ⟨URL, S, IdPDomain, /loginInfo, ⟨⟩⟩
57:        let s'.refXHR := Random()
58:        let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
59:        let s'.phase := expectLoginState
60:     end if
61:  case expectLoginState:
62:     let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
63:     let input := CHOOSEINPUT(scriptinputs, pattern)
64:     if input ≠ null then
65:        if input.Body ≡ Logged then
66:           let user ∈ ids
67:           let Url := ⟨URL, S, IdPDomain, /login, ⟨⟩⟩  mystates'.refXHR := Random()
68:           let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨credential, CredentialofUser(user, secret)⟩⟩, s'.refXHR
69:           let s'.phase := expectLoginResult
70:        else if input.Body ≡ Unlogged then
71:           let PID_RP := s'.Parameters[PID_RP]
72:           let Enpt := s'.Parameters[Enpt]
73:           let Nonce := s'.Parameters[Nonce]
74:           let Url := ⟨URL, S, IdPDomain, /authorize,
                  ⟨⟨PID_RP, PID_RP⟩, ⟨Enpt, Enpt⟩, ⟨Nonce, Nonce⟩⟩⟩
75:           let s'.refXHR := Random()
76:           let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
77:           let s'.phase := expectToken
78:        end if
```

```
    end if
  case expectLoginResult:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≠ null then
      if input.Body ≠ LoginSuccess then
        let stop ⟨⟩
      end if
      let PID_RP := s'.Parameters[PID_RP]
      let Enpt := s'.Parameters[Enpt]
      let Nonce := s'.Parameters[Nonce]
      let Url := ⟨URL, S, IdPDomain, /authorize,
        ⟨⟨PID_RP, PID_RP⟩, ⟨Enpt, Enpt⟩, ⟨Nonce, Nonce⟩⟩⟩
      let s'.refXHR := Random()
      let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
      let s'.phase := expectToken
    end if
  case expectToken:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≠ null then
      let Token := input.Body[Token]
      let RPOringin := ⟨s'.Parameters[Enpt_RP], S⟩
      let command := ⟨POSTMESSAGE, target, ⟨Token, Token⟩, RPOrigin⟩
      let s.phase := stop
    end if
end switch
let stop ⟨s', cookies, command⟩
```

B.4 RP script process

The RP script process is only invoked to process the events, (a) self-triggering events for opening the new window; (b) the postMessage from other scripts for PID_{RP} negotiation, posting PID_{RP}-registration result and identity token; (c) the HTTP response for downloading the RP *Cert*, retrieving identity token request and confirming login result. The detailed procedure of dealing with these events is shown as follows.

4 RP_Script_Relation

```
Input: ⟨tree, docID, scriptstate, scriptinputs, cookies, ids, secret⟩
let s' := scriptstate
let command := ⟨⟩
let IdPWindow := SUBWINDOW(tree, docnonce).ID_w
let RPDomain := s'.RPDomain
let IdPOringin := ⟨s'.IdPDomian, S⟩
switch s'.phase do
  case start:
    let Url := ⟨URL, S, RPDomain, /login, ⟨⟩⟩
    let command := ⟨IFRAME, Url, _SELF⟩
    let s'.phase := expectt
  case expectt:
    let pattern := ⟨POSTMESSAGE, *, Content, *⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≠ null then
      let t := input.Content[t]
      let Url := ⟨URL, S, RPDomain, /startNegotiation, ⟨⟩⟩
```

```
17:         let s'.refXHR := Random()
18:         let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨t, t⟩⟩, s'.refXHR⟩
19:         let s'.phase := expectCert
20:       end if
21:     case expectCert:
22:       let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
23:       let input := CHOOSEINPUT(scriptinputs, pattern)
24:       if input ≠ null then
25:         let Cert := input.Content[Cert]
26:         let command := ⟨POSTMESSAGE, IdPWindow, ⟨⟨Cert, Cert⟩⟩, IdPOringin⟩
27:         let s'.phase := expectRegistrationResult
28:       end if
29:     case expectRegistrationResult:
30:       let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
31:       let input := CHOOSEINPUT(scriptinputs, pattern)
32:       if input ≠ null then
33:         let RegistrationResult := input.Content[RegistrationResult]
34:         let Url := ⟨URL, S, RPDomain, /registrationResult, ⟨⟩⟩
35:         let s'.refXHR := Random()
36:         let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, s'.refXHR⟩
37:         let s'.phase := expectTokenRequest
38:       end if
39:     case expectTokenRequest:
40:       let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
41:       let input := CHOOSEINPUT(scriptinputs, pattern)
42:       if input ≠ null then
43:         let PID_RP := input.Content.Body[PID_RP]
44:         let Enpt := input.Content.Body[Enpt]
45:         let Nonce := input.Content.Body[Nonce]
46:         let command := ⟨POSTMESSAGE, IdPWindow,
              ⟨⟨PID_RP, PID_RP⟩, ⟨Enpt, Enpt⟩, ⟨Nonce, Nonce⟩⟩, IdPOringin⟩
47:         let s'.phase := expectToken
48:       end if
49:     case expectToken:
50:       let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
51:       let input := CHOOSEINPUT(scriptinputs, pattern)
52:       if input ≠ null then
53:         let Token := input.Content[Token]
54:         let Url := ⟨URL, S, RPDomain, /uploadToken, ⟨⟩⟩
55:         let s'.refXHR := Random()
56:         let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨Token, Token⟩⟩, s'.refXHR⟩
57:         let s'.phase := expectLoginResult
58:       end if
59:     case expectLoginResult:
60:       let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
61:       let input := CHOOSEINPUT(scriptinputs, pattern)
62:       if input ≠ null then
63:         if input.Body ≡ LoginSuccess then
64:           let Load Homepage
65:         end if
66:       end if
67: end switch
```