

# UPPRESSO: An Unlinkable Privacy-PREserving Single Sign-On System

## Abstract

As a widely adopted identity management and authentication mechanism in today's Internet, single sign-on (SSO) allows a user to maintain only the credential for the identity provider (IdP), instead of one credential for each relying party (RP), which shifts the burden of user authentication from RPs to the IdP. However, SSO introduces new privacy leakage threats, since (a) a curious IdP could track *all* the RPs a user has visited, and (b) collusive RPs could learn a user's online profile by linking her identifiers and activities across multiple RPs. Several privacy-preserving SSO solutions have been proposed to defend against either the curious IdP or collusive RPs, however, none of them can address both 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. We first formally analyze the privacy dilemma between SSO security requirements and the new privacy requirements, and convert the SSO privacy problem into an identifier-transformation problem. Then, we design a novel *transformed RP designation* scheme to transform the identifier of the RP, to which the user requests to log in, into a privacy-preserving pseudo-identifier ( $PID_{RP}$ ) through the cooperation between the user and the RP. Our *trapdoor user identification* scheme allows the RP to obtain a trapdoor from the transformation process and use it to derive a unique account of the user at that RP from her privacy-preserving pseudo-identifier ( $PID_U$ ) generated by the IdP. The login process of UPPRESSO follows the service pattern of OpenID Connect (OIDC), a widely deployed SSO system, with minimum modifications. And the system is platform independent. Our analysis shows UPPRESSO provides a comprehensive privacy protection while achieving the same security guarantees of OIDC.

## 1 Introduction

Single sign-on (SSO) systems such as OpenID Connect (OIDC) [1], OAuth [2] and SAML [3] have been widely deployed in today's Internet for identity management and au-

thentication. With SSO, a user can log in to a website, referred to as the relying party (RP), using her account registered at another website, known as the identity provider (IdP). An RP delegates user authentication to a trusted IdP, who then generates identity proofs for the users visiting the RP. Therefore, the user needs to remember only one credential at the IdP, instead of maintaining multiple credentials for different RPs.

However, SSO systems have been continuously found vulnerable and insecure [4–13]. Besides, the adoption of SSO raises public concern about user privacy [14–17], in the fear that an adversary may be able to track to which RP(s) the user has logged in. *Unfortunately, almost all the existing SSO protocols leak user privacy in different ways.* Take a widely used SSO protocol, OIDC, as an example. As shown in Fig. 2, the login process starts when a user sends a login request to the RP, who then constructs a request for identity proof with its identity and redirects the request to the IdP. After authenticating the user, the IdP generates an identity proof with the identities of both the user and RP, which is returned to the user and forwarded to the RP. Finally, the RP verifies the identity proof to decide if the user is allowed to log in. In such login instances, by design, an IdP can always see when and where a user attempts to log in, to generate the identity proof. As a result, a curious IdP can easily track the RPs that a target user has visited over time. This data can be further analyzed to profile users' online activities, as in other web tracking attacks. So, we call this privacy attack *IdP-based login tracing*, which has also been reported by previous research [16, 17]. Similarly, by design, the RPs can learn users' identities from the received identity proofs. If the IdP uses a unique or related user identifier(s) to generate identity proofs for her to access different RPs [18, 19], curious RPs can collude to link the login requests, associate her attributes across multiple RPs, and track her online activities [14]. We denote this privacy risk as *RP-based identity linkage*.

As SSO becomes a popular safeguard for various privacy-sensitive web services, the privacy concern is considered more prominent and severe than it was in the past. Recently, many IdPs have taken user privacy in SSO into serious consider-

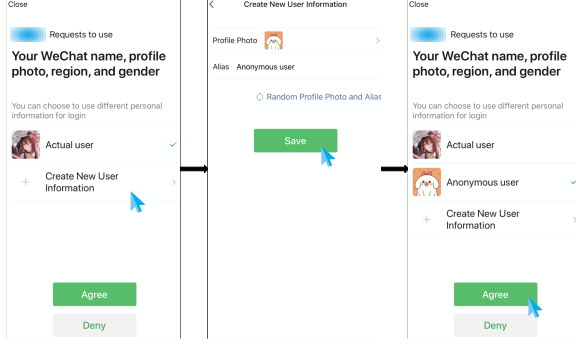


Figure 1: Anonymous SSO login in WeChat.

ation and offered new protection against RP-based identity linkage. For example, Figure 1 shows the IdP service provided by WeChat, one of the most popular instant messaging applications in China. It allows a user to create new accounts with new profile attributes to log in to different RPs. Similarly, IdPs such as NORDIC APIS and CURITY suggest adopting pairwise pseudonymous identifier (PPID) in SSO [20, 21], and Active Directory Federation Services [22] and Oracle Access Management [23] support the use of PPID.

However, these efforts cannot protect users from IdP-based login tracing. While privacy-savvy users may provide few personal information to web applications to avoid user tracking or profiling, the use of popular SSO services such as Google Account opens a door for IdPs and application providers to recover users' online traces and profiles, which makes users' privacy protection effort in vain. Several large IdPs, especially the social IdPs, are known to be interested in collecting users' online behavioral data for various purposes (e.g., Screenwise Meter [24] and Onavo [25]). Serving the IdP role makes it possible for them to collect such information. Meanwhile, service providers hosting multiple web applications take an advantaged position to correlate users' multiple logins at different RPs through internal information integration. Finally, privacy-preserving record linkage [26] and private set intersection [27] technologies allow multiple RPs to share data without violating the privacy policies, which pave the path for cross-organizational RP-based identity linkage.

Several solutions have been proposed to protect user privacy in SSO login [14–17]. However, to the best of our knowledge, *none of them provides a comprehensive protection against IdP-based login tracing and RP-based identity linkage at the same time* (we provide a brief review of existing solutions and their limitations in Section 2.2). Moreover, the techniques proposed so far to defend each of the two attacks cannot be directly integrated, since they require modifications to the SSO flows that are essentially conflicting and cannot occur simultaneously. This requires a non-trivial re-design of the SSO system to defend against both attacks, while providing a secure and compatible SSO service.

In this paper, we are among the first to conceptualize the privacy problem in SSO as an *identifier transformation prob-*

*lem* and explain the reasons that limit existing solutions from fully protecting user privacy against curious IdPs and collusive RPs. Based on our analysis, we propose an Unlinkable Privacy-PREserving Single Sign-On (UPPRESSO) system for comprehensive protection against both privacy attacks. In particular, UPPRESSO designs three one-way identifier-transformation functions based on the elliptic curve cryptography. The first two convert the identities of the user and RP into privacy-preserving pseudo-identifiers using a randomly chosen trapdoor  $T$ . This prevents IdP-based login tracing by hiding the RPs' identities from the IdP and RP-based identity linkage by hiding the user's identity from the RPs. Meanwhile, an RP can link multiple login sessions of the same user by converting her different pseudo-identifiers to an invariant user account using a special identifier-transformation function and the trapdoor  $T$ , without knowing the user's real identity. Finally, we formally prove that the accounts of the same user at different RPs are independent and cannot be correlated by collusive RPs (in Section 6). We summarize our contributions as follows.

- We formalize the SSO privacy problems as an identifier-transformation problem and analyze the limitations of existing privacy solutions for SSO.
- We propose a comprehensive solution to hide the users' login traces from curious IdPs and collusive RPs. To the best of our knowledge, UPPRESSO is the first SSO system that secures SSO services against IdP-based login tracing and RP-based identity linkage.
- We formally prove the privacy of UPPRESSO based on the DDH assumption [28] and the security of UPPRESSO based on a formal model of the web infrastructure. Our results show that UPPRESSO provides expected and satisfying security and privacy protection to SSO.
- UPPRESSO is compatible with existing SSO systems, since it requires only small modifications to support new identifier-transformation functions. We implement a prototype of UPPRESSO based on an open-source implementation of OIDC, which leverages HTML 5 features in the implementation and thus can be used across platforms (e.g., PCs, smartphones, and other devices).
- We compare the performance of our UPPRESSO prototype with the state-of-the-art SSO systems (i.e., OIDC [1] and SPRESSO [17]) and demonstrate its efficiency.

The rest of the paper is organized as follows. We introduce the background in Section 2, the privacy problem and our design ideas in Section 3, and the threat model in Section 4. Section 5 describes the detailed design of UPPRESSO based on identifier transformation, followed by a formal analysis of the privacy and security in Sections 6 and 7. We explain the implementation specifics and experimental evaluation in Section 8, discuss the extensions and related works in Sections 9 and 10, and conclude our work in Section 11.

## 2 Background

In this section, we first introduce a popular SSO protocol, OIDC, to explain typical SSO login flows. Then, we discuss existing SSO privacy solutions and their limitations.

### 2.1 OpenID Connect

OIDC is one of the most popular SSO protocols [1]. It involves three entities, i.e., *users*, the *identity provider (IdP)*, and *relying parties (RPs)*. Users and RPs register at the IdP with identifiers and other necessary information such as credentials and RP endpoints (e.g., the URLs to receive identity proofs). The IdP should maintain these attributes securely.

**OIDC Implicit 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). UPPRESSO is compatible with all three flows. For brevity, we will present our design and implementation based on the OIDC implicit flow in the rest of the paper and discuss the extension to support the authorization code flow in Section 9.

As shown in Figure 2, first, the user initiates a login request to an RP. Then, the RP constructs an identity proof request with its identifier, an endpoint to receive the identity proof and the scope of requested user attributes. The RP sends the request to the user who will redirect it to the IdP. If the user has not been authenticated yet, the IdP initiates an authentication process to authenticate the user based on her identity and credential. If a privacy-preserving pseudo-identifier is used, this process also involves mapping the real identifier ( $ID_U$ ) to the pseudo identifier ( $PID_U$ ). Once successfully authenticating the user, the IdP generates an identity proof (called *id token*) and returns it to the RP endpoint through user redirection. The id token contains a user identifier (either  $ID_U$  or  $PID_U$ ), an RP identifier ( $ID_{RP}$ ), the issuer, a validity period, the requested user attributes, etc. If the RP endpoint has not been registered at the IdP, the IdP will return a warning to notify the user about potential identity proof leakage. Besides redirecting the messages between the RP and the IdP, the user also checks if the RP is permitted to obtain the user attributes in the identity proof. Usually, the redirection and checking actions are handled by a user-controlled software called *user agent* (e.g., browser). Finally, the RP verifies the received identity proof and makes the authentication decision.

**RP Dynamic Registration.** OIDC also supports *RP dynamic registration* [29]. When an RP first registers at an IdP, it obtains a registration token with which the RP can update its information (e.g., endpoints) with the IdP at a later time. After a successful dynamic registration, the RP obtains a new  $ID_{RP}$  from the IdP. UPPRESSO leverages this function and slightly modifies the dynamic registration process to implement the *PID<sub>RP</sub> registration* process (see details in Section 5.C), which allows an RP to generate different privacy-preserving RP identifiers and register them with the IdP.

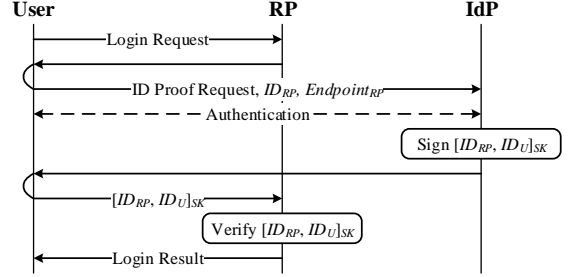


Figure 2: The implicit flow of OIDC.

### 2.2 Existing Privacy Solutions for SSO

Pairwise pseudonymous identifier (PPID) is recommended by NIST [15] and specified in several SSO protocols [1, 30] to hide the real user identity from the RPs. It can be generated by the IdP to identify a user to an RP. PPIDs for one user at different RPs should not be correlated with each other, so that collusive RPs cannot link a user’s logins. However, PPID-based approaches cannot prevent IdP-based login tracing, since the IdP needs to know which RP the user visits to generate the correct identify proof.

On the contrary, BrowserID [16] and SPRESSO [17] were proposed to defend against IdP-based login tracing. However, both solutions are vulnerable to RP-based identity linkage. In BrowserID (and its prototypes known as Mozilla Persona [31] and Firefox Accounts [19]), the IdP does not know the identity of the requesting RP. Instead, it generates a special “identity proof” to bind the user’s unique identifier (e.g., email address) to a public key, so that the user can sign another subsidiary identity proof to bind her identity with the RP’s identity and send both identity proofs to the RP. Obviously, when a user logs in to different RPs, the RPs can extract the same user identifier from different identity proofs and correlate these logins. In SPRESSO, the RP creates a one-time pseudo-identifier for itself at each login. Then, the IdP generates an identity proof binding this pseudo-identifier of the RP and the user’s identity (i.e., email address). Similarly, the RPs can correlate a user’s logins using her unique identifier in the identity proofs.

## 3 The Privacy Dilemma and UPPRESSO Overview

Next, we overview the required security and privacy properties of an SSO system. Then, we conceptualize the SSO privacy problem as an identifier-transformation problem and explain the privacy dilemma behind existing solutions. Finally, we present the design goals of UPPRESSO. We list the notations used in the discussion in Table 1 for reference.

### 3.1 Security Properties of SSO

The primary goal of SSO services is to support secure user authentication [17], which ensures that a legitimate user can always log in to (her unique account at) an honest RP. To achieve this, the identity proof generated by the IdP should

Table 1: The notations used in UPPRESSO.

Notation	Definition	Attribute
$q$	A large prime, the size of the underlying field.	Long-term
$E$	An elliptic curve defined over a finite field $\mathbb{F}_q$ (e.g., P-256).	Long-term
$G$	A point in $E(\mathbb{F}_q)$ , known as the base point.	Long-term
$n$	the order of the base point $G$ .	Long-term
$SK, PK$	The private/public key of IdP.	Long-term
$ID_{RP}$	An RP's unique identity, a point in cyclic subgroup of $E(\mathbb{F}_q)$ generated base on $P$ .	Long-term
$Cert_{RP}$	An RP certificate, containing the RP's identity and end-point.	Long-term
$ID_U$	A user's unique identity.	Long-term
$Account$	A user's identifier at an RP: $A = ID_U ID_{RP}$ (denoted as $A$ in equations).	Long-term
$PID_{RP}$	$PID_{RP} = N_U ID_{RP}$ , an RP's pseudo-identifier.	One-time
$PID_U$	$PID_U = ID_U PID_{RP}$ , a user's pseudo-identifier.	One-time
$N_U$	A user-generated nonce for $PID_{RP}$ .	One-time
$T$	The trapdoor to derive $Account$ : $T = N_U^{-1} \bmod n$ .	One-time

explicitly specify the user who is authenticated by the IdP (i.e., **user identification**) and the RP to which the user requests to log in (i.e., **RP designation**). To provide continuous service, the user identification property also requires an RP to be able to recognize a user and correlate her multiple logins by her unique identifier or account. Moreover, the identify proof generated by the IdP should be transmitted only to the dedicated RP (directly or through the user) (i.e., **confidentiality**) and should not be modified or forged (i.e., **integrity**). We summarize these four security properties from theoretical analysis of SSO designs [32–34] and practical attacks [4–13, 35–42].

Many attacks exploit vulnerabilities in SSO design and implementation to break at least one of the four security properties. The adversary mainly aims to log in to an honest RP as a victim user (called *impersonation attacks*) or allure a victim user to log in to an honest RP under the attacker's account (called *identity injection attacks*). For example, Friendcaster used to accept every received identity proof (i.e., a violation of RP designation) [40]. So, a malicious RP can replay a received identity proof to Friendcaster and log in as the victim user. If identity proofs are leaked (i.e., a violation of confidentiality) [4, 5, 7–9], the adversary can directly impersonate the victim user. It was also reported that some RPs of Google ID SSO accepted user attributes that were not tied to the identity proofs (i.e., a violation of integrity) [4], so that an adversary can insert arbitrary attributes (e.g., email of the adversary or another user) into the identity proof of the victim user.

### 3.2 The Privacy Dilemma in SSO Identity Proofs

A secure SSO system should have *all* four security properties discussed above while preventing IdP-based login tracing and RP-based identity linkage privacy leakage. However, meeting the security and privacy requirements at the same time incurs a dilemma in the generation of identity proofs.

An identity proof contains identities/identifiers of a user and an RP, which is to tell the RP that this user has been authenticated by the IdP. Since the IdP always knows the user's identity (i.e.,  $ID_U$ ), to prevent IdP-based login tracing, we should not reveal the RP's long-term identity (i.e.,  $ID_{RP}$ )

to the IdP. This leaves us no option but to use a transitional pseudo-identifier (denoted as  $PID_{RP}$ ) to hide the real  $ID_{RP}$ . To ensure RP designation,  $PID_{RP}$  should be uniquely associated with the RP in the identity proof request, while an RP should use different  $PID_{RPs}$  in different login instances.  $PID_{RP}$  can be generated by the user, the RP, or together. But it should be computationally infeasible for the IdP to derive the  $ID_{RP}$  from a  $PID_{RP}$ .

Meanwhile, to prevent RP-based identity linkage, the IdP should not directly include  $ID_U$  in the identity proof. So, the IdP has to generate a transitional pseudo-identifier for the user (denoted as  $PID_U$ ) and bind it to the identity proof. While  $PID_U$  should not disclose any information for the RP to derive  $ID_U$ , it should also allow the RP to distinguish one user from other users. This means, to ensure user identification, an RP should be able to correlate one user's different  $PID_U$ s in different login instances, for example, by linking them to the same unique user account, as the RP recognizes a user by her user account (denoted as  $Account$ ). However, collusive RPs should not be able to correlate one user's  $Accounts$  at different RPs or infer some accounts belong to the same user.

We illustrate the relationships among the identities, pseudo-identifiers and the identity proofs in Figure 3. The red and green blocks represent long-term identities and one-time pseudo-identifiers, respectively. The arrows denote how the pseudo-identifiers are obtained. To protect user privacy, the identity proof should use only one-time pseudo-identifiers and  $PID_U$  and  $PID_{RP}$  should satisfy the above requirements. This causes a **privacy dilemma** for the IdP: given a user ( $ID_U$ ) and an unknown RP ( $PID_{RP}$ ), the IdP is expected to generate a pseudo-identifier ( $PID_U$ ), which is correlated with a long-term account that uniquely identifies the user at that RP, *without knowing anything about the RP's identity nor the user account at the RP*.

To solve the dilemma, we should provide the IdP some information related to the user's  $Account$  at the RP to assist the generation of  $PID_U$ , so that  $PID_U$  can be correctly correlated with the  $Account$ . Meanwhile, such information should not provide any additional knowledge for the IdP to derive the RP's identity, or for two RPs to correlate two  $Accounts$  belonging to the same user. Therefore, the privacy protection problem can be converted into an identifier-transformation problem, which aims to design three identifier-transformation functions  $\mathcal{F}_{ID_U \mapsto PID_U}$ ,  $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ , and  $\mathcal{F}_{PID_U \mapsto Account}$  to compute  $PID_U$ ,  $PID_{RP}$ , and  $Account$ , which have the above desired properties.

### 3.3 The Identifier-Transformation Framework of UPPRESSO

To achieve this goal, UPPRESSO constructs three transformation functions in an integrated way to support *transformed RP designation* and *trapdoor user identification*, where (a) different  $PID_U$ s and  $PID_{RPs}$  are dynamically generated in different logins; (b) in each login session,  $PID_{RP}$  is used to



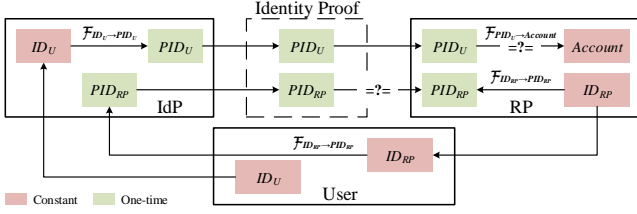


Figure 3: Identifier transformations in privacy-preserving SSO.

Table 2: Identifier-transformation in privacy-preserving SSO.

Solution	$PID_U$	$PID_{RP}$	Account
PPID	$\mathcal{F}(ID_U, ID_{RP})$	$ID_{RP}$	$PID_U$
SPRESSO	$ID_U$	$Enc(ID_{RP}    nonce)$	$ID_U$
BrowserID <sup>†</sup>	$ID_U$	$\perp$	$ID_U$
UPPRESSO	$\mathcal{F}(ID_U, PID_{RP})$	$\mathcal{F}(ID_{RP}, T)$	$\mathcal{F}(PID_U, T)$

<sup>†</sup>: BrowserID binds null  $PID_{RP}$  in the identity proofs by the IdP, but  $ID_{RP}$  is bound in the *subsidiary* identity proof signed by the user.

assist the generation of  $PID_U$ , which helps to link  $PID_U$  to *Account* using the trapdoor of this login session.

**Transformed RP designation.** To prevent IdP-based login tracing, the RP includes a  $PID_{RP}$  dynamically transformed from  $ID_{RP}$  in the identity proof request. UPPRESSO designs a novel trapdoor-based transformation function  $\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}}(ID_{RP}, T)$  to compute  $PID_{RP}$  based on  $ID_{RP}$  and a random trapdoor  $T$  that is dynamically negotiated between the user and the RP in each login. Then, the user assists the RP to register  $PID_{RP}$  at the IdP through OIDC dynamic registration. When an RP receives an identity proof, it verifies if the enclosed  $PID_{RP}$  is transformed from its  $ID_{RP}$  and the trapdoor of this login session.

**Trapdoor user identification.** To prevent RP-based identity linkage, we design a transformation function  $\mathcal{F}_{ID_U \rightarrow PID_U}(ID_U, PID_{RP})$  for the IdP to generate  $PID_U$ . When an RP receives an identity proof, another transformation function  $\mathcal{F}_{PID_U \rightarrow Account}(PID_U, PID_{RP}, T)$  is designed to help the RP to derive the *Account* from  $PID_U$  and  $PID_{RP}$  using the trapdoor it holds. Intuitively, the trapdoor  $T$  plays a role in the generations of  $PID_{RP}$  and  $PID_U$ , directly or indirectly.

### 3.4 Existing Privacy-Preserving SSO Solutions

We map three existing privacy-preserving SSO approaches (PPID [1], BrowserID [16] and SPRESSO [17]) to the identifier transformation framework in Figure 3 and summarize their potential privacy issues in Table 2. It is worth noting that when  $PID_U = ID_U$  and  $PID_{RP} = ID_{RP}$ , this framework depicts the basic SSO services with no privacy protection.

In PPID approaches, PPIDs are used as  $PID_U$ s for a user at different RPs. There are deterministic one-to-many mappings from  $ID_U$  to  $PID_U$ s. However, the RP sends its real identifier to IdP in each login and thus suffers from IdP-based login tracing.

In SPRESSO, the RP generates  $PID_{RP}$  by encrypting  $ID_{RP}$  padded with a nonce for each login session. But the IdP issues the id token using the real user identifier  $ID_U$ .

Identity proofs in BrowserID include  $ID_U$  but no RP information (i.e.,  $PID_{RP} = \perp$ ). Therefore, it suffers from RP-based identity linkage, but can prevent IdP-based login tracing.

In summary, none of the three approaches can defend against IdP-based login tracing and RP-based identity linkage at the same time. This is because in each approach, three transformation functions  $\mathcal{F}_{ID_U \rightarrow PID_U}$ ,  $\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}}$  and  $\mathcal{F}_{PID_U \rightarrow Account}$  are designed arbitrarily and function separately, which causes either  $PID_{RP} = ID_{RP}$  or  $Account = ID_U$ .

## 4 Threat Model and Assumptions

### 4.1 Threat Model

In UPPRESSO, we consider the IdP is curious-but-honest, while some users and RPs could be compromised by adversaries. Malicious users and RPs may behave arbitrarily or collude with each other, attempting to break the security and privacy guarantees for benign users.

**Curious-but-honest IdP.** A curious-but-honest IdP strictly follows the protocol, while being interested in learning user privacy. For example, it may store all the received messages to infer the relationship among  $ID_U$ ,  $ID_{RP}$ ,  $PID_U$ , and  $PID_{RP}$  to trace a user's login activities at multiple RPs. We also assume the IdP is well-protected. For example, the IdP is trusted to maintain the private key for signing identity proofs and RP certificates. So, the adversaries cannot forge an identity proof or an RP certificate.

**Malicious Users.** We assume the adversary can control a set of users, for example by stealing users' credentials [43, 44] or directly registering Sybil accounts at the IdP and RPs. They may impersonate a victim user at honest RPs, or trick a victim user to log in to an honest RP under the adversary's account. For example, a malicious user may modify, insert, drop or replay a message, or deviate arbitrarily from the specifications when processing  $ID_{RP}$ ,  $PID_{RP}$ , and identity proofs.

**Malicious RPs.** The adversary can also control a set of RPs, for example, by directly registering at the IdP as an RP or exploiting software vulnerabilities to compromise some RPs. The malicious RPs may behave arbitrarily to break security and privacy guarantees. To do so, a malicious RP may manipulate its  $PID_{RP}$  to trick the users to submit identity proofs generated for an honest RP to itself. Or, it may manipulate its  $PID_{RP}$  to affect the generation of  $PID_U$  and analyze the relationship between  $PID_U$  and *Account*.

**Collusive Users and RPs.** Malicious users and RPs may collude with each other to break the security and privacy guarantees. For example, the adversary can first pretend to be an honest RP and trick the victim user into submitting her identity proof to it. With the valid identity proof, it can impersonate the victim user and log in to the honest RP.

### 4.2 Assumptions

We made a few assumptions about the information and implementation of the SSO system under study. First, we consider user attributes as distinctive and indistinctive attributes, where

distinctive attributes contain identifiable information about a user such as her telephone number, address, driver's license, etc. We assume the RPs cannot obtain distinctive attributes in an SSO login, since a privacy-savvy user is less likely to permit the RPs to access such information, or even does not register such information with the IdP at all. Therefore, the privacy leakage due to user re-identification is considered out of the scope of this work. Moreover, we focus only on privacy attacks enabled by SSO protocols, but not network attacks such as traffic analysis that trace a user's logins at different RPs from network traffic. Next, we assume the user agent deployed at honest users is correctly implemented so that it can transmit messages to the dedicated receivers as expected. Finally, we assume TLS is adopted to secure the communications between honest entities, and the cryptographic algorithms (such as RSA and SHA-256) and building blocks (such as random number generators) are correctly implemented.

## 5 The Design of UPPRESSO

After we conceptualize the privacy problem into an identifier-transformation problem, the design of UPPRESSO is mainly about designing three identifier-transformation functions to generate pseudo-identifiers for the user and RP as well as linking the user's pseudo-identifier to her account at an RP. In this section, we first present our design of these three functions to support *transformed RP designation* and *trapdoor user identification* properties, and then describe the details of the UPPRESSO system and its login flow.

### 5.1 Identifier-transformation Functions in UPPRESSO

We construct three functions,  $\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}}$ ,  $\mathcal{F}_{ID_U \rightarrow PID_U}$  and  $\mathcal{F}_{PID_U \rightarrow Account}$ , based on the NIST elliptic curve  $P-256$ , where  $G$  is a point on the curve (known as the base point) and  $n$  is the order of the base point  $G$ . The IdP assigns long-term identifiers  $ID_U$  and  $ID_{RP}$  to a user and an RP respectively, when they first register at the IdP. Without loss of generality, we assume the IdP assigns a unique random number to each user as  $ID_U$ , where  $1 < ID_U < n$ , and  $ID_{RP}$  is a point on the curve generated based on  $G$ .

**The RP Identifier Transformation Function.** In each login session, the user assists the RP to convert  $ID_{RP}$  into a pseudo-identifier  $PID_{RP}$ . In particular, the user selects a random number  $N_U$  ( $1 < N_U < n$ ) and calculates  $PID_{RP}$  as:

$$\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}} : PID_{RP} = N_U \cdot ID_{RP} \quad (1)$$

With  $\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}}$ , it is computationally infeasible for the IdP to derive  $ID_{RP}$  due to the discrete logarithm problem. Moreover, the nonce  $N_U$  ensures that: (a)  $PID_{RP}$  is a one-time pseudo-identifier in each login; and (b)  $PID_{RP}$  is dynamically generated in each login. When a user visits the same RP multiple times, different  $PID_{RPs}$  will be generated, which cannot be correlated to the same RP by the curious IdP.

**The User Identifier Transformation Function.** To avoid using the user identity  $ID_U$  in identity proofs, the IdP can

convert  $ID_U$  into a pseudo-identifier for the user as follows:

$$\mathcal{F}_{ID_U \rightarrow PID_U} : PID_U = ID_U \cdot PID_{RP} \quad (2)$$

From Equations 1 and 2, we see that  $PID_U = (N_U ID_U \bmod n) \cdot ID_{RP}$ . So,  $PID_U$  is a one-time pseudo-identifier. It is only valid in one login session and one identity proof. As expected, the RP cannot derive  $ID_U$  from  $PID_U$  due to the discrete logarithm problem, but it can associate a user's one-time pseudo-identifier ( $PID_U$ ) registered at the IdP with her long-term identifier registered at the RP (i.e., *Account*).

**The User Account Transformation Function.** We define the trapdoor  $T$  in each login session as  $T = N_U^{-1} \bmod n$ . With this trapdoor, the RP can easily derive a unique account (denoted as  $A$  in the equation) for the user as:

$$\mathcal{F}_{PID_U \rightarrow Account} : A = T \cdot PID_U \quad (3)$$

From Equations 1, 2 and 3, we can further derive:

$$A = T \cdot PID_U = (ID_U N_U N_U^{-1} \bmod n) \cdot ID_{RP} = ID_U ID_{RP}$$

This means when a user logs in to an RP multiple times, the RP can always derive the same *Account* from different  $PID_U$ s to uniquely identify the user. However, the RP cannot derive  $ID_U$  from *Account* due to the discrete logarithm problem.

With three identifier-transformation functions, UPPRESSO supports two desirable properties discussed in Section 3.3 to satisfy *all* the security and privacy requirements of an SSO.

### 5.2 UPPRESSO Procedures

**System Initialization.** UPPRESSO consists of four procedures. Firstly, IdP calls system initialization once to establish the system. In particular, the IdP generates key pair ( $SK$ ,  $PK$ ) to sign identity proofs and RP certificates. Then, the IdP keeps  $SK$  secret, while announcing  $PK$  as public parameters.

**RP Initial Registration.** Each RP calls an initial registration process once to obtain configurations. In particular, an RP registers itself at the IdP to obtain a unique identifier  $ID_{RP}$  and the corresponding RP certificate  $Cert_{RP}$  as follows: (i) The RP sends a registration request to the IdP, including the RP endpoint (e.g., URL) to receive identity proofs; (ii) The IdP generates a unique  $ID_{RP}$  and signs  $[ID_{RP}, Endpoint_{RP}, *]$  using  $SK$ , where  $*$  denotes supplementary information such as the RP's common name; then, the IdP returns  $Cert_{RP} = [ID_{RP}, Endpoint_{RP}, *]_{SK}$  to the RP, where  $[\cdot]_{SK}$  means the message is signed using  $SK$ ; (iii) The RP verifies  $Cert_{RP}$  using  $PK$  and accepts  $ID_{RP}$  and  $Cert_{RP}$  if they are valid. Note that, in UPPRESSO,  $ID_{RP}$  must be generated by the IdP but cannot be chosen by the RP.

**User registration.** UPPRESSO adopts a similar user registration process as the ones in other SSO systems. Each user registers once at the IdP to set up a unique user identifier  $ID_U$  and the corresponding user credential.

**SSO Login.** An SSO login procedure is launched when a user requests to log in to an RP. It consists of five phases, namely scripts downloading, RP identifier transformation,  $PID_{RP}$  registration, identity proof generation, and *Account* calculation, as shown in Figure 4.

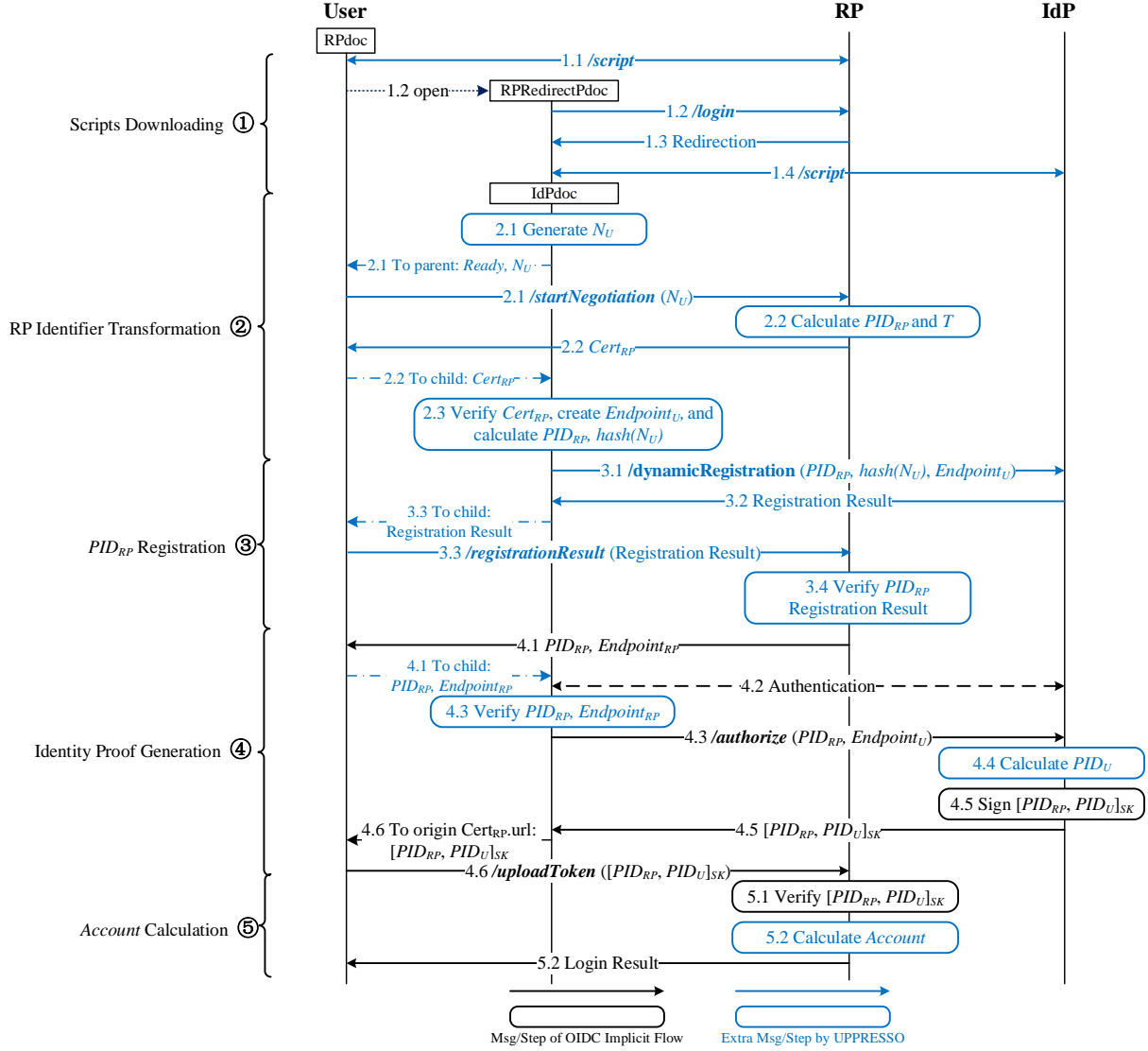


Figure 4: The flow of a user login in UPPRESSO.

1. *Scripts Downloading.* The scripts downloading phase is for the user's browser to download the scripts from the RP and IdP servers. The browser and two scripts work together to play the user agent role.

- 1.1 The user visits the RP's site to download the RP script.
- 1.2 The RP script opens a new window in the browser to visit the login path at the RP server.
- 1.3 The visit is redirected to the IdP's script site.
- 1.4 The new window visits the IdP's script site and downloads the IdP script.

2. *RP Identifier Transformation.* The user and RP cooperate to generate  $PID_{RP} = N_U ID_{RP}$ . To hide the RP's endpoint from the IdP, the user needs to create a new endpoint to replace the real endpoint of the RP.

- 2.1 The IdP script chooses a random  $N_U$  ( $1 < N_U < n$ ) and sends it to the RP script through postMessage. Then, the RP script sends  $N_U$  to the RP server.

- 2.2 The RP verifies  $N_U \neq 0 \pmod n$ , calculates  $PID_{RP}$  and derives the trapdoor  $T = N_U^{-1} \pmod n$ . To acknowledge the negotiation of  $PID_{RP}$ , the RP replies with  $Cert_{RP}$ , which is transmitted from the RP script to the IdP script through postMessage.

- 2.3 The IdP script verifies  $Cert_{RP}$ , extracts  $ID_{RP}$  from  $Cert_{RP}$  and calculates  $PID_{RP} = N_U ID_{RP}$  and  $nonce = hash(N_U)$ . It also creates a one-time endpoint for the RP. If  $Cert_{RP}$  is invalid, the user halts the negotiation.

It is important to ensure that the RP endpoint is not tampered with by the adversary. In other OIDC systems, the IdP obtains the RP endpoint from RP registration and verifies the endpoint in the identity proof request. However, the IdP in UPPRESSO sees only a one-time endpoint. So, we ask the user to verify the correctness of the RP endpoint using the RP certificate.

3.  *$PID_{RP}$  Registration.* In this phase, the user registers a new RP with  $PID_{RP}$  at the IdP using OIDC's dynamic registration.

This step has to be conducted by the user but not the RP. Otherwise, the IdP can associate  $PID_{RP}$  and  $ID_{RP}$ .

- 3.1 The IdP script sends the  $PID_{RP}$  registration request  $[PID_{RP}, Hash(N_U), Endpoint_U]$  to the IdP.
- 3.2 The IdP checks the unexpired  $PID_{RPs}$  to verify if the received  $PID_{RP}$  is unique. Then, it signs the response as  $[PID_{RP}, hash(N_U), validity]_{SK}$ , where *validity* denotes when the  $PID_{RP}$  will expire.
- 3.3 The IdP script forwards the registration result to the RP server through the RP script.
- 3.4 The RP verifies the IdP's signature and accepts the result only if  $PID_{RP}$  and  $hash(N_U)$  match those in the negotiation and  $PID_{RP}$  is not expired.

$hash(N_U)$  is a nonce to distinguish different login sessions because there is a very small chance that the same  $PID_{RP}$  is generated for two RPs even using two different pairs of  $ID_{RPs}$  and  $N_U$ s. This nonce avoids a registration result being acceptable to two RPs.

4. *ID Proof Generation.* In this phase, the IdP calculates  $PID_U = ID_U PID_{RP}$  and signs the identity proof.

- 4.1 The RP constructs an identity proof request containing its  $PID_{RP}$  and  $Endpoint_{RP}$ , which is then forwarded to the IdP script through the RP script.
- 4.2 The IdP authenticates the user if she has not been authenticated yet.
- 4.3 First, the user checks the scope of the requested attributes, while the IdP script verifies that  $PID_{RP}$  and  $Endpoint_{RP}$  in the request are valid. Then, the IdP script replaces the RP's endpoint with the newly registered one-time  $Endpoint_U$  and sends the modified identity proof request to the IdP server.
- 4.4 The IdP verifies if  $PID_{RP}$  and  $Endpoint_U$  are registered and unexpired, and then calculates  $PID_U = ID_U PID_{RP}$  for the authenticated user.
- 4.5 The IdP constructs and signs the identity proof  $[PID_{RP}, PID_U, Iss, ValTime, Attr]_{SK}$ , where *Iss* is the identifier of the IdP, *ValTime* is the validity period, and *Attr* contains the requested attributes.
- 4.6 The IdP sends the identity proof to the one-time endpoint. The IdP script forwards the identity proof to the RP script that holds the origin  $Endpoint_{RP}$ . Finally, the RP script sends it to the RP server.

In this phase, if any check fails, the process will be halted. For example, the user halts the process if  $PID_{RP}$  in the identity proof request is inconsistent with the negotiated one. The IdP rejects the identity proof request, if the pair of  $PID_{RP}$  and  $Endpoint_U$  has not been registered.

5. *Account calculation.* The RP verifies the identity proof, derives the user's unique *Account*, and allows her to log in.

- 5.1 The RP verifies the identity proof, including the signature, validity period, and if  $PID_{RP}$  is consistent with the negotiated one. If any fails, the RP rejects this login.
- 5.2 The RP extracts  $PID_U$ , calculates  $Account = T \cdot PID_U$ , and allows the user to log in.

### 5.3 Compatibility with OIDC

UPPRESSO does not introduce any new role nor change the security assumptions for each role. It follows a similar logic flow as OIDC in SSO login and only requires small modifications to perform identifier transformation. Among the five phases, the *scripts downloading* and *RP identifier transformation* phases are newly introduced by UPPRESSO. The browser is required to download two scripts from the IdP and RP and most of the designed operations in these two phases are performed by the scripts in the browser. So, we require minimal modifications to the IdP and RP servers providing new network interfaces (i.e., the new URLs for downloading resources). The other three phases adopt a similar communication pattern as OIDC. In particular, the  $PID_{RP}$  registration phase can be viewed as a variant of the RP dynamic registration flow of OIDC [29], which allows an entity to register its identity and endpoint at the IdP.

UPPRESSO can also support the authorization code flow of OIDC with small modifications (to be discussed in Section 9).

## 6 Provable Security Analysis of Privacy

In this section, we will give the privacy proof and show that UPPRESSO is secure against both IdP-based login tracing and RP-based identity linkage attacks.

**IdP-based login tracing.** As shown in figure 4, the only information that is related to the RP's identity and accessible to the IdP is  $PID_{RP}$ , which is converted from  $ID_{RP}$  using a random  $N_U$ . Since  $N_U$  is randomly chosen from  $\mathbb{Z}_n$  by the user and the IdP does not involve in the process, the IdP should treat  $PID_{RP}$  as randomly chosen from  $\mathbb{G}$ . So, the IdP cannot recognize the RP nor derive its real identity. Therefore, IdP-based identity linkage becomes impossible in UPPRESSO.

Next, we will prove that UPPRESSO prevents RP-based identity linkage based on the Decisional Diffie-Hellman (DDH) assumption [28]. Here, we briefly introduce the DDH assumption: Let  $q$  be a large prime and  $\mathbb{G}$  denote a cyclic group of order  $n$  of an elliptic curve  $E(\mathbb{F}_q)$ . Assume that  $n$  is also a large prime. Let  $P$  be a generator point of  $\mathbb{G}$ . The DDH assumption for  $\mathbb{G}$  states that for any probabilistic polynomial time (PPT) algorithm  $D$ , the two probability distributions  $\{aP, bP, abP\}$  and  $\{aP, bP, cP\}$ , where  $a, b$ , and  $c$  are randomly and independently chosen from  $\mathbb{Z}_n$ , are computationally indistinguishable in the sense that there is a negligible  $\sigma(k)$  with the security parameter  $k$  such that:

$$Pr[D(P, aP, bP, abP) = 1] - Pr[D(P, aP, bP, cP) = 1] = \sigma(k)$$

**RP-based identity linkage.** Collusive RPs can act arbitrarily to correlate  $PID_U$ s at different RPs and guess if they belong to the same user. Therefore, we consider the collusive RPs are playing a guessing *Game*. In this *Game*, the IdP and the users act as the challenger, while the collusive RPs act as the adversary (denoted as  $A$  in Figure 5). RP-based identity linkage is impossible in UPPRESSO if and only if the adversary has no advantage over the challenger in the guessing game.



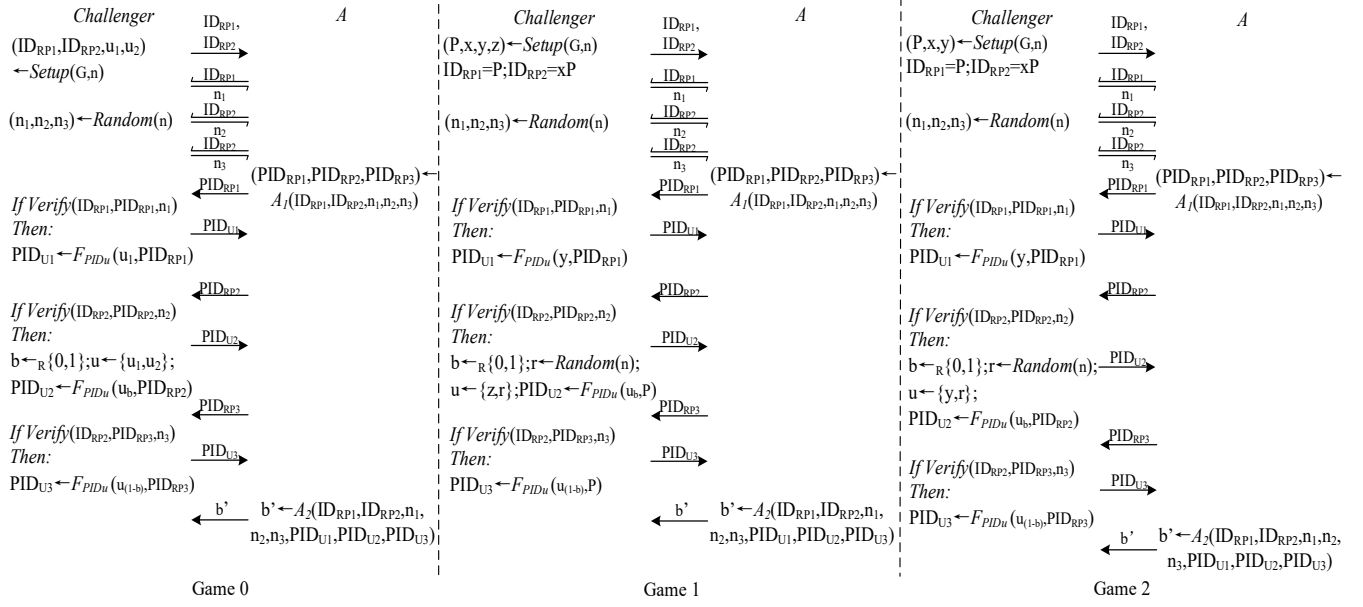


Figure 5: Game.

Next, we build the model of the guessing Game to depict the interactions between the challenger and the adversary. First, we describe the challenger's actions in the Game.

- Initialization: In the initialization phase, the challenger generates the  $ID_{RPs}$  and  $ID_{Us}$  for multiple RPs and users, based on the initialization algorithm  $\text{Setup}(G, n)$  where  $G$  and  $n$  have been already defined in Table 1.
- Random number generation: While challenger receives the  $ID_{RPs}$  from adversary, it generates random numbers  $N_U \in \mathbb{Z}_n$  for each  $ID_{RPs}$  creating  $PID_{RP}$  using algorithm  $\text{Random}(n)$ .
- $PID_U$  generation: In this phase, challenger firstly receives and verifies whether  $PID_{RP}$  is correctly generated based on  $ID_{RP}$  and corresponding  $N_U$  (i.e., the algorithm  $\text{Verify}(ID_{RP}, PID_{RP}, N_U)$ ). Then it generates the  $PID_U$  using the algorithm  $F_{PID_U}(ID_U, PID_{RP})$  and sends it to adversary.

Firstly we give the realistic model of the  $ID_U$ -guessing Game, shown as Game 0 in figure 5. In this Game, adversary firstly achieves 2  $ID_{RPs}$  and 3  $N_{Us}$  (denote as  $n_1, n_2, n_3$ ), then generates 3  $PID_{RPs}$ . In the challenger's view, these  $PID_{RPs}$  are related with 3  $N_{Us}$  respectively. Following the challenger generates  $PID_{Us}$  for different  $PID_{RPs}$  based on the 2  $ID_{Us}$  (denote as  $u_1, u_2$ ) generated in initialization phase. Then challenger generates the random  $b=0$  or 1.  $PID_{U1}$  is generated based on  $ID_{U1}$ . However  $PID_{U2}$  and  $PID_{U3}$  are generated according to  $b$ . That is,  $PID_{U2} = ID_{Ub}PID_{RP2}$ , and  $PID_{U3} = ID_{U(1-b)}PID_{RP3}$ .

Finally adversary returns the  $b'$ . While  $b = b'$  is true, we consider the adversary succeeds. We define the event  $[b' = b]$  in Game 0 as  $\Gamma$ . The probability  $Pr[\Gamma]$  must be 1/2 as while the adversary cannot take any advantage of guessing  $b$  (i.e.

whether  $PID_{Us}$  belong to same  $ID_U$ ). Therefore, it is proved that UPRESSO is protected from RP-based identity linkage only if  $Pr[\Gamma] = 1/2$ .

Then we build the ideal model of Game, of which the probability that an adversary succeeds in guessing  $b$  is 1/2. The model is shown as Game 1 in Figure 5. We use  $z$  and  $r$  for generating  $PID_{U2}$  and  $PID_{U3}$  instead of  $ID_{U1}$  and  $ID_{U2}$ . As  $z$  and  $r$  are randomly generated and unknown to  $A$ , the adversary has no knowledge about  $b$ . We define the event  $[b' = b]$  in Game 1 as  $\Gamma_1$ .  $Pr[\Gamma_1]$  must be equal to 1/2.

Therefore, we only need to prove that  $|Pr[\Gamma_1] - Pr[\Gamma]| = \sigma(n)$ , where  $\sigma(n)$  is negligible. Here we give another model of Game, shown as Game 2 in Figure 5. We can find that in fact it only sets the exact values of the parameters defined in Game 0, such as  $ID_{RP2} = xID_{RP1}$ ,  $u_1 = y$  and  $u_2 = r$  ( $r$  is a random number). Here we define the event  $[b' = b]$  in Game 2 as  $\Gamma_2$ . There should be  $Pr[\Gamma_2] = Pr[\Gamma]$ .

We are going to prove that  $|Pr[\Gamma_1] - Pr[\Gamma_2]| = \sigma(n)$ . In each Games, adversary derives the  $b'$  from collected data,  $\{ID_{RP1}, ID_{RP2}, n_1, n_2, n_3, PID_{U1}, PID_{U2}, PID_{U3}\}$ , with the algorithm  $A_2$ . Now we replace the parameters in Game 1 and Game 2 with exact values. In Game 1,  $b'_{game1} \leftarrow A_2(P, xP, n_1, n_2, n_3, y, n_1P, zP, rP)$ , and in Game 2,  $b'_{game2} \leftarrow A_2(P, xP, n_1, n_2, n_3, y, n_1P, x, y, n_2P, r, n_3P)$  or  $b'_{game2} \leftarrow A_2(P, xP, n_1, n_2, n_3, y, n_1P, r, n_2P, x, y, n_3P)$ .  $n_1, n_2, n_3$  are randomly chosen by challenger, which provides no extra information about  $ID_U$  and can be eliminated by adversary. Therefore, there is, in Game 1,  $b'_{game1} \leftarrow A_2(P, xP, yP, zP, rP)$ , and in Game 2,  $b'_{game2} \leftarrow A_2(P, xP, yP, xyP, xrP)$ . As  $r$  is randomly chosen and unknown to adversary, both  $rP$  and  $xrP$  are random points, there is  $b'_{game1} \leftarrow A_2(P, xP, yP, zP, r_1P)$  and  $b'_{game2} \leftarrow A_2(P, xP, yP, xyP, r_2P)$ . It can be proved that no non-negligible difference value between the success probability in

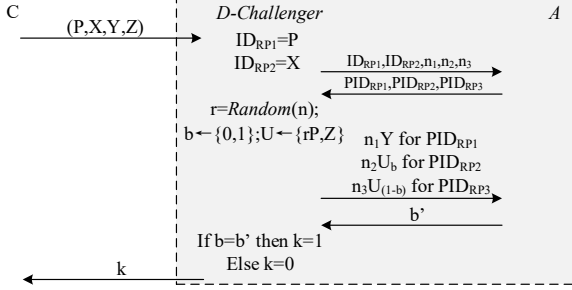


Figure 6: Distinguishing algorithm.

Game 1 and Game 2 according to DDH assumption. Otherwise, we can build the PPT distinguishing algorithm breaking DDH assumption based on the adversary.

**Distinguishing algorithm.** The distinguishing algorithm  $D$  is shown as Figure 6. The inputs of the algorithm is  $\{P, X, Y, Z\}$ . While the input is in the form  $\{P, xP, yP, zP\}_{x,y,z \in \mathbb{Z}_n}$ , for the adversary it is the Game 1. As the input is  $\{P, xP, yP, xyP\}_{x,y \in \mathbb{Z}_n}$ , for the adversary it is the Game 2. So, there is,

$$Pr[D(P, xP, yP, zP) = 1] = Pr[\Gamma_1]$$

$$Pr[D(P, xP, yP, xyP) = 1] = Pr[\Gamma_2]$$

Therefore,  $|Pr[\Gamma_1] - Pr[\Gamma_2]| = \sigma(n)$ , where  $\sigma(n)$  is negligible, and  $n$  is the security parameter. So, the adversary has no advantage to guessing  $b$  in Game 0, that means he cannot distinguish whether two  $PID_U$ s for different RPs belong to the same user or not.

The RP-based identity linkage attack is not possible in UPPRESSO.

## 7 Security Analysis

We formally analyze security properties of UPPRESSO based on the Dolev-Yao style web model [17], which has been widely used in the formal analysis of SSO protocols such as OAuth 2.0 [33] and OIDC [34]. For brevity, we focus on the modifications introduced by UPPRESSO in this paper and neglect the proofs for the security of DNS and HTTPS requests. We refer interested readers to [17] for details.

### 7.1 The Web Model

The Dolev-Yao model abstracts the entities in a system, such as browsers and web servers, as *atomic processes*, which communicate with each other through the *events*. [17] also defines *scripting processes* to model client-side scripting such as JavaScript, so a web system consists of a set of atomic and scripting processes. The state of a system, called a *configuration*, consists of the current states of all atomic processes and all the events that can be accepted by these processes. We list the definitions of these notations as below [17].

*Messages* are the basic data carriers among web nodes, such as HTTP requests and responses.

*Events* are the basic communication elements in the model. An event contains the addresses of sender and receiver, and a message.

**Atomic Processes.** An *atomic Dolev-Yao (DY) process* is a tuple  $p = (I^p, Z^p, R^p, s_0^p)$ , where  $I^p$  is the set of addresses that the process listens to,  $Z^p$  is the set of states (i.e., terms) that describes the process,  $s_0^p$  is an initial state, and  $R^p$  is the mapping from an input state  $s \in Z^p$  and an event  $e$  to a new state  $s'$  and an event  $e'$ . Each atomic process also contains a set of nonces that it may use.

**Scripting Processes** represent client-side scripts loaded by the browser to provide server-defined functions to the browser. However, a scripting process must rely on an atomic process, such as the browser, and provide the relation  $R$  called by this atomic process.

**Web system.** We can represent the web infrastructure as a web system of form  $(\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ , where  $\mathcal{W}$  is the set of atomic processes containing both honest and malicious processes,  $\mathcal{S}$  is the set of scripting processes including honest and malicious scripts,  $\text{script}$  is the set of concrete script codes related to specific scripting processes in  $\mathcal{S}$ , and  $E^0$  is the set of events acceptable to the processes in  $\mathcal{W}$ .

A *configuration* of this web system is a tuple  $(S, E, N)$ , where  $S$  is the current states of all processes in  $\mathcal{W}$ ,  $E$  is the set of events that the processes accept, and  $N$  is a global sequence of nonces that have not been used by the processes yet.

A *run step* is the system migrating from configurations  $(S, E, N)$  to  $(S', E', N')$  by processing an event  $e \in E$ .

**The Formal Model of UPPRESSO.** Accordingly, we model UPPRESSO as a web system, which is defined as  $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \text{script}, E^0)$ .  $\mathcal{W}$  is a finite set of atomic processes in UPPRESSO, which contains an IdP server process, a finite set of web servers for the honest RPs, a finite set of honest browsers, and a finite set of attacker processes. Here, we consider all the RP processes and browser processes are honest, and model an RP or a browser controlled by an adversary as an atomic attacker process.  $\mathcal{S}$  is a finite set of scripting processes, which contains  $\text{script}_{rp}$ ,  $\text{script}_{idp}$  and  $\text{script}_{attacker}$ , where  $\text{script}_{rp}$  and  $\text{script}_{idp}$  are honest scripts downloaded by an RP process and the IdP process, and  $\text{script}_{attacker}$  denotes a script downloaded by an attacker process that exists in all browser processes. The details about each process are described in Appendix.

### 7.2 Security of UPPRESSO

At the very beginning, it is to be emphasized that, whether an RP accepted *Account* can be controlled by an adversary may be the most noticeable question for readers. For example, the adversary may try to make the conflict  $\text{Account}_1 = ID_{U_1} ID_{RP_1}$ ,  $\text{Account}_2 = ID_{U_2} ID_{RP_2}$ , and  $\text{Account}_1 = \text{Account}_2$  possible, where  $ID_{U_1}$  and  $ID_{RP_1}$  belong to the honest user and RP. Here we give the direct conclusion and brief proofs in advance. **The  $PID_U$  achieved by an adversary cannot be transformed into the honest user's  $Account$  at an honest RP.**

**Proof.** While the  $PID_{U_{adv}}$  is achieved by the adversary, it must be in the form  $PID_{U_{adv}} = (a) ID_{U_{adv}} PID_{RP_{adv}}; (b) ID_U PID_{RP_{adv}}; \text{ or } (c) ID_{U_{adv}} PID_{RP}$ . Due to the  $PID_{RP}$  reg-

istration, the (a) and (b) cannot be accepted by honest RP. In (c) the  $PID_{RP}$  must be created as  $N_U ID_{RP_{hon}}$ , while the  $hash(N_U)$  is included in registration result. Moreover, the trap-door  $T = N_U^{-1} \bmod n$  cannot be controlled by adversary. So, the  $PID_{U_{adv}}$  must be transformed into  $ID_{U_{adv}} ID_{RP_{hon}}$  (i.e., the *Account*). Therefore, the *Account* cannot be controlled by the adversary.

In the security analysis, we consider web systems  $\mathcal{UWS}$  defined in Section 7.1. In this model, we consider only network attacker can control (malicious) browsers and RPs. The analysis of the security of UPPRESSO is to prove the below theorem:

**Theorem 1.** *Let  $\mathcal{UWS}$  be a UPPRESSO web system defined above. Then,  $\mathcal{UWS}$  is secure.*

In Section 3.1, we describe the fundamental security properties that an SSO system should satisfy with. The adversary should not succeed in *impersonation attacks* or *identity injection attacks*. We consider the visits to an RP's resource paths are controlled by the visitors' cookie. So, if such a system exists, the attacker could break the security only if he owns the cookie bound to the honest user. Therefore, we define a secure UPPRESSO as below.

**Definition 1.** Let  $\mathcal{UWS}$  be a UPPRESSO web system.  $\mathcal{UWS}$  is secure *iff* any authenticated cookie  $c(u, r)$  of an honest user  $u$  for an honest RP  $r \in \mathcal{W}$  is unknown to the attacker  $a$ .

To prove that an attacker  $a$  does not know the authenticated cookie  $c(u, r)$ , we want to show that (A)  $a$  cannot obtain any  $c(u, r)$  owned by  $u$ ; (B) if  $c$  is an unauthenticated cookie owned by  $a$ ,  $c$  cannot be set as  $c(u, r)$ , i.e., being authenticated by  $r$  for  $u$ ; and (C) an honest user  $u$  should not use the attacker's cookie to visit an honest RP (i.e.,  $c(a, r)$ ).

$\mathcal{UWS}$  meeting the requirement (A) can be proved by the following Lemma.

**Lemma 1.** The cookie owned by an honest user cannot be leaked to the attacker.

First, due to the same-origin policy, an honest browser should not leak the cookie to any attacker. Based on the UPPRESSO model, we also prove that the RP server and the RP script will not send any cookie to other processes. Therefore, the attackers cannot obtain the  $u$ 's authenticated cookie.

Next, to prove  $\mathcal{UWS}$  satisfies the requirement (B), we define the process that authenticates a cookie as below.

**Definition 2.** In  $\mathcal{UWS}$ , a cookie  $c$  is set as an authenticated cookie  $c(u, r)$  for a user  $u$  and an RP  $r$  only when  $r$  receives a valid identity proof of  $u$  from the owner of  $c$ .

**Lemma 2.** In  $\mathcal{UWS}$ , an attacker cannot obtain the password of an honest user  $u$ .

**Lemma 3.** In  $\mathcal{UWS}$ , an attacker cannot forge or modify the proofs issued by the IdP.

Lemma 2 can be easily proved because the password is only sent by an honest IdP scripting process to the IdP server. Lemma 3 can be proved by showing that the proofs issued by the IdP process are signed and verified. With Lemma 2 and

Lemma 3, we can prove the following lemma.

**Lemma 4.** In  $\mathcal{UWS}$ , an attacker cannot obtain a valid identity proof for an honest user  $u$ .

Here, we provide a brief proof for Lemma 4. According to the proof as the beginning of this section, it is proved that the *Account* accepted as  $u$  by RP must be issued for  $u$ . Moreover, an honest user  $u$  only sends the identity proof from the IdP scripting process to the receiver specified by the RP certificate  $cert_r$ . Therefore, an adversary cannot achieve a valid identity proof.

Then, we prove a  $\mathcal{UWS}$  system meets the requirement (C). We here propose another lemma.

**Lemma 5.** The attacker cannot know a valid  $PID_{RP}$  negotiated by a user  $u$  and an RP  $r$ .

In UPPRESSO, an RP shares different  $PID_{RPs}$  with different users. According to this lemma, the adversary cannot forge the token accepted by RP in honest user's login session. Therefore, requirement (C) is satisfied.

Finally, we prove  $\mathcal{UWS}$  satisfies requirements (A) and (B) in Definition 1. As a result, Theorem 1 is proved. Due to space limit, we include all the detailed proofs of the lemmas and theorems in the Appendix.

## 8 Implementation and Evaluation

We have implemented the UPPRESSO prototype, and evaluated its performance by comparing with the original OIDC which only prevents RP-based identity linkage, and SPRESSO which only prevents IdP-based login tracing.

### 8.1 Implementation

We adopt SHA-256 for digest generation, and RSA-2048 for signature generation. We choose the NIST elliptic curve  $P-256$  to create  $ID_{RP}$  (the point generated based on base point  $G$ ),  $N_U$  and  $ID_U \in \mathbb{Z}_n$  ( $n$  is the order of  $G$ ). UPPRESSO includes the processing at the IdP, users and the RPs. The implementations at each entity are as follows.

The implementation of the IdP only needs small modifications on the existing OIDC implementation. The UPPRESSO IdP is implemented based on MITREid Connect [45], an open-source OIDC Java implementation certificated by the OpenID Foundation [46]. We add 3 lines of Java code to calculate  $PID_U$ , about 20 lines to modify the way to send identity proof to the RP, about 50 lines to the function of dynamic registration to support  $PID_{RP}$  registration, i.e., checking  $PID_{RP}$  and adding a signature and validity period in the response. The calculations of  $ID_{RP}$ ,  $PID_U$  and RSA signature are implemented based on Java built-in cryptographic libraries (e.g., BigInteger).

The user-side processing is implemented as a JavaScript code provided by IdP and RP server, respectively containing about 200 lines and 150 lines of codes, to provide the functions in Steps 2.1, 2.3 and 4.3. The cryptographic computations, e.g.,  $Cert_{RP}$  verification and  $PID_{RP}$  negotiation, are implemented based on jsrsasn [47], an efficient JavaScript

cryptographic library.

We provide a Java SDK for RPs to integrate UPPRESSO. The SDK provides 2 functions to encapsulate RP’s processings: one for RP identifier transformation,  $PID_{RP}$  registration and identity proof request generation; while the other for identity proof verification and  $Account$  calculation. The SDK is implemented based on the Spring Boot framework with about 1000 lines code, and cryptographic computations are implemented based on Spring Security library. An RP only needs to invoke these two functions for the integration.

## 8.2 Performance Evaluation

**Environment.** The evaluation was performed on 3 machines, one (3.4GHz CPU, 8GB RAM, 500GB SSD, Windows 10) as IdP, one (3.1GHz CPU, 8GB RAM, 128GB SSD, Windows 10) as an RP, and the last one (2.9GHz CPU, 8GB RAM, 128GB SSD, Windows 10) as a user. The user agent is Chrome v75.0.3770.100. And the machines are connected by an isolated 1Gbps network.

**Setting.** We compare UPPRESSO with MITREid Connect [45] and SPRESSO [17], where MITREid Connect provides open-source Java implementations [45] of IdP and RP’s SDK, and SPRESSO provides the JavaScript implementations based on node.js for all entities [17]. We implemented a Java RP based on Spring Boot framework for UPPRESSO and MITREid Connect, by integrating the corresponding SDK respectively. The RPs in all the three schemes provide the same function, i.e., extracting the user’s account from the identity proof. We have measured the time for a user’s login at an RP, and calculated the average values of 1000 measurements.

For better analysis, we divide a login into 3 phases according to the lifecycle of identity proof: **Identity proof requesting** (Steps 1.1-4.3 in Figure 4), the RP (and user) constructing and transmitting the request to IdP; **Identity proof generation** (Steps 4.4-4.6 in Figure 4), the IdP generating identity proof (no user authentication); and **Identity proof acceptance** (Steps 4.5-5.2 in Figure 4), the RP server receives, verifies and parse the identity proof relayed from the IdP;

**Results.** The evaluation results are provided in Figure 7. The overall processing times are 113 ms, 308 ms and 310 ms for MITREid Connect, SPRESSO and UPPRESSO, respectively. The details are as follows. The significant overhead in UPPRESSO is opening the new window and downloading the script from IdP, which needs about 104 ms. This overhead could be reduced by implicitly conducting this procedure when the user visits the RP website.

In the requesting, UPPRESSO requires 271 ms totally. The significant overhead in UPPRESSO is opening the new window and downloading the script from IdP, which needs about 104 ms. This overhead could be reduced by implicitly conducting this procedure when the user visits the RP website. SPRESSO needs 19 ms for the RP to obtain IdP’s public key and encrypt its domain, and MITREid Connect only needs 10 ms.

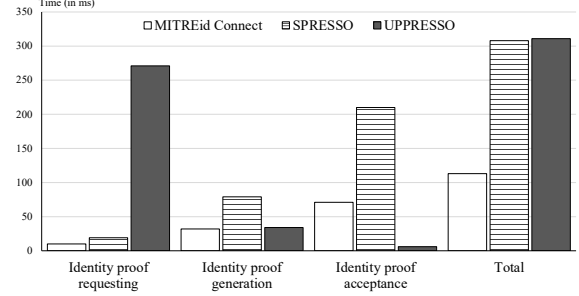


Figure 7: The Evaluation.

In the generation, UPPRESSO needs totally 34 ms including computing  $PID_U$ , compared to MITREid Connect which only needs 32 ms. SPRESSO requires 71 ms, as it implements the IdP based on node.js and therefore can only adopt a JavaScript cryptographic library, while others adopt a more efficient Java library.

In the identity proof acceptance, UPPRESSO only needs about 6 ms. MITREid Connect requires the IdP to send the identity proof to the RP’s web page which then sends the proof to the RP server through a JavaScript function, and needs 71 ms. SPRESSO needs the longest time (210 ms) due the complicated processing at the user’s browser.

## 9 Discussions and Future Work

In this section, we discuss some related issues and our future work.

**OIDC authorization code flow support.** The privacy-preserving functions  $\mathcal{F}_{ID_U \rightarrow PID_U}$ ,  $\mathcal{F}_{ID_{RP} \rightarrow PID_{RP}}$  and  $\mathcal{F}_{PID_U \rightarrow Account}$  can be integrated into OIDC authorization code flow directly, therefore RP-based identity linkage and IdP-based login tracing are still prevented during the construction and parsing of identity proof. The only privacy leakage is introduced by the transmission, as RP servers obtain the identity proof directly from the IdP in this flow, which allows the IdP to obtain RP’s network information (e.g., IP address). UPPRESSO needs to integrate existing anonymous networks (e.g., Tor) to prevent this leakage.

**Malicious IdP mitigation.** The IdP is assumed to assign a unique  $ID_{RP}$  in  $Cert_{RP}$  for each RP and generate the correct  $PID_U$  for each login. The malicious IdP may attempt to provide the incorrect  $ID_{RP}$  and  $PID_U$ , which could be prevented by integrating certificate transparency [48] and user’s identifier check [17]. With certificate transparency [48], the monitors check the uniqueness of  $ID_{RP}$  among all the certificates stored in the log server. To prevent the malicious IdP from injecting any incorrect  $PID_U$ , the user could provide a nickname to the RP for an extra check as in SPRESSO [17].

**Identity linkage through cookie.** In UPPRESSO, IdP does not provide any distinctive information (such as  $ID_U$ ) of each user to RP, which avoids RP-based identity linkage. However, the cookie of each user may be exploited by RPs to correlate the same user. For instance, while the user has logged in to  $RP_A$  with  $Account_A$ , RP may redirect user’s  $Account_A$  to  $RP_B$



through the hidden iframe. That is, as long as the user has logged in to  $RP_B$  with  $Account_B$ , the user would be correlated. Moreover, this attack is not only appeared in SSO systems, but also existed in all user-account systems. However, this attack can be easily detected through multiple methods, such as checking the iframe in script, observing redirection flow through browser network tool, and detecting the redirection based on the browser extension.

## 10 Related Works

Various SSO protocols have been proposed, such as, OIDC, OAuth 2.0, SAML, Central Authentication Service (CAS) [49] and Kerberos [50]. These protocols are widely adopted in Google, Facebook, Shibboleth project [51], Java applications and etc. And, plenty of works have been conducted on privacy protection and security analysis for SSO systems.

**Privacy-preserving SSO systems.** As suggested by NIST [15], SSO systems should prevent both RP-based identity linkage and IdP-based login tracing. The pairwise user identifier is adopted in SAML [3] and OIDC [1], and only prevents RP-based identity linkage; while SPRESSO [17] and BrowserID [16] only prevent IdP-based login tracing. BrowserID is adopted in Persona [31] and Firefox Accounts [19], however an analysis on Persona found IdP-based login tracing could still succeed [16, 52]. UPPRESSO prevents both the RP-based identity linkage and IdP-based login tracing, and could be integrated into OIDC which has been formally analyzed [34].

**Anonymous SSO systems.** Anonymous SSO schemes are designed to allow users to access a service (i.e. RP) protected by a verifier (i.e., IdP) without revealing their identities. One of the earliest anonymous SSO systems was proposed for Global System for Mobile (GSM) communication in 2008 [53]. The notion of anonymous SSO was formalized [54] in 2013. And, various cryptographic primitives, such as group signature, zero-knowledge proof and etc., were adopted to design anonymous SSO schemes [54, 55]. Anonymous SSO schemes are designed for the anonymous services, and not applicable to common services which need user identification.

**Formal analysis on SSO standards.** The SSO standards (e.g., SAML, OAuth and OIDC) have been formally analyzed. Fett et al. [33, 34] have conducted the formal analysis on OAuth 2.0 and OIDC standards based on an expressive Dolev-Yao style model [52], and proposed two new attacks, i.e., 307 redirect attack and IdP Mix-Up attack. When the IdP misuses HTTP 307 status code for redirection, the sensitive information (e.g., credentials) entered at the IdP will be leaked to the RP by the user's browser. While, IdP Mix-Up attack confuses the RP about which IdP is used and makes the victim RP send the identity proof to the malicious IdP, which breaks the confidentiality of the identity proof. Fett et al. [33, 34] have proved that OAuth 2.0 and OIDC are secure once these two attacks prevented. UPPRESSO could be integrated into OIDC, which

simplifies its security analysis. [32] formally analyzed SAML and its variant proposed by Google, and found that Google's variant of SAML doesn't set RP's identifier in the identity proof, which breaks RP designation.

**Analysis on SSO implementations.** Various vulnerabilities were found in SSO implementations, and then exploited for impersonation and identity injection attacks by breaking the confidentiality [4, 5, 7–9], integrity [4, 6, 9–12] or RP designation [9–13] of identity proof. Wang et al. [4] analyzed the SSO implementations of Google and Facebook from the view of the browser relayed traffic, and found logic flaws in IdPs and RPs to break the confidentiality and integrity of identity proof. An authentication flaw was found in Google Apps [7], allowing a malicious RP to hijack a user's authentication attempt and inject the malicious code to steal the cookie (or identity proof) for the targeted RP, breaking the confidentiality. The integrity has been tampered with in SAML, OAuth and OIDC systems [4, 6, 10–12], due to various vulnerabilities, such as XML Signature wrapping (XSW) [6], RP's incomplete verification [4, 10, 12], IdP spoofing [11, 12] and etc. And, a dedicated, bidirectional authenticated secure channel was proposed to improve the confidentiality and integrity of identity proof [41]. The vulnerabilities were also found to break the RP designation, such as the incorrect binding at IdPs [10, 13], insufficient verification at RPs [11–13]. Automatical tools, such as SSOScan [35], OAuthTester [37] and S3KVetter [13], have been designed to detect vulnerabilities for breaking the confidentiality, integrity or RP designation of identity proof.

## 11 Conclusion

In this paper, we propose UPPRESSO, an unlinkable privacy-preserving single sign-on system, which protects a user's login activities at different RPs against both curious IdP and collusive RPs. To the best of our knowledge, UPPRESSO is the first approach that defend against both IdP-based login tracing and RP-based identity linkage privacy threats at the same time. To achieve these goals, we convert the privacy problem in SSO services into an identifier-transformation problem and design three transformation functions based on elliptic curve cryptography, where  $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$  prevents curious IdP from knowing RP's identity,  $\mathcal{F}_{ID_U \mapsto PID_U}$  prevents collusive RPs from linking a user based on her identifier, and  $\mathcal{F}_{PID_U \mapsto Account}$  allows RP to derive an identical account for a user in her multiple logins. The three functions could be integrated with existing SSO protocols, such as OIDC, to enhance the protection of user privacy, without breaking any security guarantee of SSO. Moreover, the evaluation on the prototype of UPPRESSO demonstrates that it supports an efficient SSO service, where a single login takes only 310 ms on average.

## References

- [1] Nat Sakimura, John Bradley, Mike Jones, Breno de Medeiros, and Chuck Mortimore, "OpenID Con-

- nect core 1.0 incorporating errata set 1,” *The OpenID Foundation, specification*, 2014.
- [2] Dick Hardt, “The OAuth 2.0 Authorization Framework,” *RFC*, vol. 6749, pp. 1–76, 2012.
  - [3] John Hughes, Scott Cantor, Jeff Hodges, Frederick Hirsch, Prateek Mishra, Rob Philpott, and Eve Maler, “Profiles for the OASIS Security Assertion Markup Language (SAML) v2.0,” *OASIS standard*, 2005, Accessed August 20, 2019.
  - [4] Rui Wang, Shuo Chen, and XiaoFeng Wang, “Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, California, USA, 2012, pp. 365–379.
  - [5] San-Tsai Sun and Konstantin Beznosov, “The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems,” in *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA, 2012, pp. 378–390.
  - [6] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen, “On breaking SAML: Be whoever you want to be,” in *Proceedings of the 21th USENIX Security Symposium*, Bellevue, WA, USA, 2012, pp. 397–412.
  - [7] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti, “An authentication flaw in browser-based single sign-on protocols: Impact and remediations,” *Computers & Security*, vol. 33, pp. 41–58, 2013.
  - [8] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffei, “Discovering concrete attacks on website authorization by formal analysis,” *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.
  - [9] Wanpeng Li and Chris J. Mitchell, “Analysing the security of Google’s implementation of OpenID Connect,” in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, San Sebastián, Spain, 2016, pp. 357–376.
  - [10] Hui Wang, Yuanyuan Zhang, Juanru Li, and Dawu Gu, “The achilles heel of OAuth: A multi-platform study of OAuth-based authentication,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, Los Angeles, CA, USA, 2016, pp. 167–176.
  - [11] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk, “Do not trust me: Using malicious IdPs for analyzing and attacking single sign-on,” in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrücken, Germany, 2016, pp. 321–336.
  - [12] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich, “Sok: Single sign-on security - an evaluation of OpenID Connect,” in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, Paris, France, 2017, pp. 251–266.
  - [13] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang, “Vetting single sign-on SDK implementations via symbolic reasoning,” in *Proceedings of the 27th USENIX Security Symposium*, USENIX Security, Baltimore, MD, USA, 2018, pp. 1459–1474.
  - [14] Eve Maler and Drummond Reed, “The venn of identity: Options and issues in federated identity management,” *IEEE Security & Privacy*, vol. 6, no. 2, pp. 16–23, 2008.
  - [15] Paul A Grassi, M Garcia, and J Fenton, “NIST special publication 800-63c digital identity guidelines: Federation and assertions,” *National Institute of Standards and Technology*, Los Altos, CA, 2017.
  - [16] Daniel Fett, Ralf Küsters, and Guido Schmitz, “Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web,” in *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*, 2015, pp. 43–65.
  - [17] Daniel Fett, Ralf Küsters, and Guido Schmitz, “SPRESSO: A secure, privacy-respecting single sign-on system for the web,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, CO, USA, 2015, pp. 1358–1369.
  - [18] “Google identity platform,” <https://developers.google.com/identity/>, Accessed August 20, 2019.
  - [19] “About firefox accounts,” <https://mozilla.github.io/application-services/docs/accounts/welcome.html>, Accessed August 20, 2019.
  - [20] “Build gdpr compliant apis with openid connect,” <https://nordicapis.com/build-gdpr-compliant-apis-with-openid-connect/>, Accessed August 20, 2020.
  - [21] “Pairwise pseudonymous identifiers,” <https://curity.io/resources/architect/openid-connect/ppid-intro/>, Accessed August 20, 2020.

- [22] “The role of claims,” <https://docs.microsoft.com/en-us/windows-server/identity/ad-fs/technical-reference/the-role-of-claims>, Accessed August 20, 2020.
- [23] “Introducing identity federation in oracle access management,” [https://docs.oracle.com/cd/E40329\\_01/admin.1112/e27239/oif\\_1.htm](https://docs.oracle.com/cd/E40329_01/admin.1112/e27239/oif_1.htm), Accessed August 20, 2020.
- [24] Sydney Li and Jason Kelley, “Google screenwise: An unwise trade of all your privacy for cash,” <https://www.eff.org/deeplinks/2019/02/google-screenwise-unwise-trade-all-your-privacy-cash>, Accessed July 20, 2019.
- [25] Bennett Cyphers and Jason Kelley, “What we should learn from “facebook research”,” <https://www.eff.org/deeplinks/2019/01/what-we-should-learn-facebook-research>, Accessed July 20, 2019.
- [26] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant, “Information sharing across private databases,” in *Proceedings of the 30th ACM International Conference on Management of Data (SIGMOD)*, San Diego, California, USA, 2003, pp. 86–97.
- [27] Emiliano De Cristofaro and Gene Tsudik, “Practical private set intersection protocols with linear complexity,” in *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC)*, Canary Islands, Spain, 2010, pp. 143–159.
- [28] Shafi Goldwasser and Yael Tauman Kalai, “Cryptographic assumptions: A position paper,” in *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, Eyal Kushilevitz and Tal Malkin, Eds. 2016, vol. 9562 of *Lecture Notes in Computer Science*, pp. 505–522, Springer.
- [29] Nat Sakimura, John Bradley, Mike Jones, Breno de Medeiros, and Chuck Mortimore, “OpenID Connect dynamic client registration 1.0 incorporating errata set 1,” *The OpenID Foundation, specification*, 2014.
- [30] Thomas Hardjono and Scott Cantor, “SAML v2.0 subject identifier attributes profile version 1.0,” *OASIS standard*, 2019.
- [31] Mozilla Developer Network (MDN), “Persona,” <https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona>.
- [32] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra, “Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps,” in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, Alexandria, VA, USA, 2008, pp. 1–10.
- [33] Daniel Fett, Ralf Küsters, and Guido Schmitz, “A comprehensive formal security analysis of OAuth 2.0,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016, pp. 1204–1215.
- [34] Daniel Fett, Ralf Küsters, and Guido Schmitz, “The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines,” in *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*, Santa Barbara, CA, USA, 2017, pp. 189–202.
- [35] Yuchen Zhou and David Evans, “SSOScan: Automated testing of web applications for single sign-on vulnerabilities,” in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, USA, 2014, pp. 495–510.
- [36] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu, “Vulnerability assessment of OAuth implementations in Android applications,” in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, USA, 2015, pp. 61–70.
- [37] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu, “Model-based security testing: An empirical study on OAuth 2.0 implementations,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, Xi’an, China, 2016, pp. 651–662.
- [38] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi, “Breaking and fixing mobile app authentication with OAuth2.0-based protocols,” in *Proceedings of the 15th International Conference on Applied Cryptography and Network Security (ACNS)*, Kanazawa, Japan, 2017, pp. 313–335.
- [39] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau, “MoSSOT: An automated blackbox tester for single sign-on vulnerabilities in mobile applications,” in *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, Auckland, New Zealand, 2019, pp. 269–282.
- [40] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague, “OAuth demystified for mobile application developers,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, USA, 2014, pp. 892–903.

- [41] 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 *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Gothenburg, Sweden., 2014, pp. 276–298.
- [42] Mohamed Shehab and Fadi Mohsen, “Towards enhancing the security of OAuth implementations in smart phones,” in *Proceedings of the 3rd IEEE International Conference on Mobile Services, Anchorage, AK, USA*, 2014, pp. 39–46.
- [43] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang, “Targeted online password guessing: An underestimated threat,” in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016, pp. 1242–1254.
- [44] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin, “oPass: A user authentication protocol resistant to password stealing and password reuse attacks,” *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 7, no. 2, pp. 651–663, 2012.
- [45] “MITREid connect /openid-connect-java-spring-server,” <https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server>, Accessed August 20, 2019.
- [46] “Openid foundation,” <https://openid.net/certification/>, Accessed August 20, 2019.
- [47] “jsrsasign,” <https://kjur.github.io/jsrsasign/>, Accessed August 20, 2019.
- [48] Ben Laurie, Adam Langley, and Emilia Käsper, “Certificate transparency,” *RFC*, vol. 6962, pp. 1–27, 2013.
- [49] Pascal Aubry, Vincent Mathieu, and Julien Marchal, “ESUP-portail: Open source single sign-on with CAS (central authentication service),” *Proceedings of EUNIS04–IT Innovation in a Changing World*, pp. 172–178, 2004.
- [50] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller, “Kerberos: An authentication service for open network systems,” in *Proceedings of the USENIX Winter Conference. Dallas, Texas, USA*, 1988, pp. 191–202.
- [51] “The shibboleth project,” <https://www.shibboleth.net>, Accessed July 30, 2019.
- [52] Daniel Fett, Ralf Küsters, and Guido Schmitz, “An expressive model for the web infrastructure: Definition and application to the BrowserID SSO system,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, Berkeley, CA, USA, 2014, pp. 673–688.
- [53] Kalid Elmufti, Dasun Weerasinghe, Muttukrishnan Rajarajan, and Veselin Rakocevic, “Anonymous authentication for mobile single sign-on to protect user privacy,” *International Journal of Mobile Communications (IJMC)*, vol. 6, no. 6, pp. 760–769, 2008.
- [54] Jingquan Wang, Guilin Wang, and Willy Susilo, “Anonymous single sign-on schemes transformed from group signatures,” in *Proceedings of the 5th International Conference on Intelligent Networking and Collaborative Systems, Xi’an, China*, 2013, pp. 560–567.
- [55] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer, “Anonymous single-sign-on for n designated services with traceability,” in *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, Barcelona, Spain, 2018, pp. 470–490.

## A Web Model

### A.1 Data Format

Here we provide the details of the format of the messages we use to construct the UPPRESSO model.

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

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

An HTTP response message is the term of the form

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

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, such as 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  $\langle \langle \text{name}, \text{value} \rangle, \langle \text{name}, \text{value} \rangle, \dots \rangle$ , for example, the *parameters* in the url `http://www.example.com?type=confirm` is  $\langle \langle \text{type}, \text{confirm} \rangle \rangle$ .



- *headers* is the header content of each HTTP messages as the form  $\langle \langle name, value \rangle, \langle name, value \rangle, \dots \rangle$ , 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  $\langle \langle name, value \rangle, \langle name, value \rangle, \dots \rangle$ .
- *status* is the HTTP status code defined by HTTP standard.

**URL.** URL is a term  $\langle URL, protocol, host, path, parameters \rangle$ , 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 stands for the specific domain used by the HTTP CORS policy, where *host* and *protocol* are the same as in URL.

**POSTMESSAGE.** PostMessage is used in the browser for transmitting messages between scripts from different origins. We define the postMessage as the form  $\langle POSTMESSAGE, target, Content, Origin \rangle$ , 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's origin.

**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 in the form  $\langle XMLHTTPREQUEST, URL, methods, Body, nonce \rangle$  can be converted into HTTP request message by the browser, and  $\langle XMLHTTPREQUEST, Body, nonce \rangle$  is converted from HTTP response message.

**Data Operation.** The data used in UPPRESSO are defined in the following forms:

- **Standardized Data** is the data in the fixed format, for instance, the HTTP request is the standardized data in the form  $\langle HTTPReq, nonce, method, host, path, parameters, headers, body \rangle$ . We assume there is an HTTP request  $r := \langle HTTPReq, n, GET, example.com, /path, \langle \rangle, \langle \rangle, \langle \rangle \rangle$ , here we define the operation on the  $r$ . That is, the elements in  $r$  can be accessed in the form  $r.name$ , such that  $r.method \equiv GET$ ,  $r.path \equiv /path$  and  $r.body \equiv \langle \rangle$ .
- **Dictionary Data** is the data in the form  $\langle \langle name, value \rangle, \langle name, value \rangle, \dots \rangle$ , 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  $\langle \langle username, alice \rangle, \langle password, 123 \rangle, \langle age, 18 \rangle \rangle$ .

**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  $\langle a, b, c \rangle$ .

## A.2 Browser Model

In UPPRESSO, we assume that the browsers are honest, therefore, we only need to analyze how the browsers interactive with the scripts.

We firstly introduce the windows and documents of the browser model.

**Window.** A window  $w$  is a term of the form  $w = \langle nonce, documents, opener \rangle$ , representing the concrete browser window in the system. The *nonce* 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 window in 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  $w.opener \equiv d.nonce$ .

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

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

where document is the HTML content in the window. The *nonce* 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 scripting process downloaded from each servers. *scriptstate* is define by the script, different in each scripts. The *scriptinputs* is the message transmitted into the scripting process. The *subwindows* is the set of *nonce* of document's created windows. *active* represents whether this document is active or not.

A scripting 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. Here we give the description of the form of input and output.

- **Scripting Message Input.** The input is the term in the form

$$\langle tree, docnonce, scriptstate, stateinputs, cookies, localStorage, sessionStorage, ids, secret \rangle$$

- **Scripting Message Output.** The output is the term in the form

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

The *tree* is the relations of the opened windows and documents, which are visible to this script. *Docnonce* is the document nonce. The *Scriptstate* is a term of the form defined by each script. *Scriptinputs* is the message transmitted to script. However, the *scriptinputs* is defined as standardized forms, for example, *postMessage* is one of the forms of *scriptinputs*. *Cookies* is the set of cookies that belong to the document's origin. *LocalStorage* is the storage space for browser and *sessionStorage* is the space for each HTTP sessions. *Ids* is the set of user IDs while *secret* is the password to corresponding user ID. The *command* is the operation which is to be conducted by the browser. Here we only introduce the form of commands used in UPPRESSO system. We have defined the *postMessage* and *XMLHttpRequest* (for HTTP request) message which are the *commands*. Moreover, a term in the form  $\langle \text{IFRAME}, \text{URL}, \text{WindowNonce} \rangle$  asks the browser to create this document's subwindow and it visits the server with the URL.

### A.3 Model of UPPRESSO

In this section, we introduce the model of processes in UPPRESSO system, including IdP server process, RP server process, IdP scripting process and RP scripting process. We will focus on the state form and relation  $R$ . They can describe that what kind of event can be accepted by the process in each state, and the content of new output events and states.

#### A.3.1 IdP Server Process

The state of IdP server process is a term in the form  $\langle \text{ID}, \text{SignKey}, \text{sessions}, \text{users}, \text{RPs}, \text{Validity}, \text{Tokens} \rangle$ . Other data stored at IdP but not used during SSO authentication are not mentioned here.

- *ID* is the identifier of IdP.
- *SignKey* is the private key used by IdP to generate signatures.
- *sessions* is the term in the form of  $\langle \langle \text{Cookie}, \text{session} \rangle \rangle$ , the Cookie uniquely identifies the session and sessions store the browser uploaded messages.
- *users* is the set of user's information, including *username*, *password*, *ID<sub>U</sub>* and other attributes.
- *RPs* is the set of RP information which consists of ID of RP (*PID<sub>RP</sub>*), *Endpoints* (i.e., the set of RP's validity endpoints) and *Validity*.
- *Validity* is the validity for IdP generated signatures.
- *Tokens* is the set of IdP generated Identity proofs.

To make the description clearer, we also provide the *functions* to define the complicated procedure.

- *SecretOfID(u)* is used to search the user  $u$ 's password.

- *UIDOfUser(u)* is used to search the user  $u$ 's *ID<sub>U</sub>*.
- *ListOfPID()* is the set of IDs of registered RP.
- *EndpointsOfRP(r)* is the set of endpoints registered by the RP with ID  $r$ .
- *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 Algorithm 1 in Appendix B.

#### A.3.2 RP process

The state of RP server process is a term in the form  $\langle \text{ID}_{\text{RP}}, \text{Endpoints}, \text{IdP}, \text{Cert}, \text{sessions}, \text{users} \rangle$ . Other attributes are not mentioned here.

- *ID<sub>RP</sub>* and *Endpoints* are RP's registered information at IdP.
- *Cert* is the IdP signed RP information containing *ID<sub>RP</sub>*, *Endpoints* and other attributes.
- *IdP* is the term of the form  $\langle \text{ScriptUrl}, q, \text{PubKey} \rangle$ , where *ScriptUrl* is the site to download IdP script,  $q$  is the large prime defined before, and *PubKey* is the public key used to verify the IdP signed messages.
- *sessions* is same as it in IdP process.
- *users* is the set of users registered at this RP, each user is uniquely identified by the *Account*.

The new *functions* are defined as follows

- *ExEU(a, q)* is the Extended Euclidean algorithm, which calculates  $a^{-1} \bmod q$ .
- *Random()* generates a fresh random number.
- *RegisterUser(Account)* add the new user with *Account* into RP's user list.

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

#### A.3.3 IdP scripting process

The state of IdP scripting process *scriptstate* is a term in the form  $\langle \text{IdPDomain}, \text{Parameters}, q, \text{refXHR} \rangle$ , where

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

The new *functions* are defined as follows.

- **PARENTWINDOW**(*tree, docnonce*). The first parameter is the input relation tree defined before, and the second parameter is the nonce of a document. The output returned by the function is the current window's opener's nonce (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.

The relation of IdP scripting process *script\_idp* is shown in Appendix B Algorithm 3.

#### A.3.4 RP scripting process

The state of RP scripting process *scriptstate* is a term in the form  $\langle IdPDomain, RPDomain, Parameters, q, refXHR \rangle$ . The *RPDomain* is the host string of the corresponding RP server, and other terms are defined in the same way as in IdP scripting process.

Here, we define the function **SUBWINDOW**(*tree, docnonce*), which takes the *tree* defined above and the current document's *nonce* as the input. And it selects the *nonce* of the first window opened by this document as the output. However, if there is no opened windows, it returns null.

The relation of RP scripting process *script\_rp* is shown in Appendix B Algorithm 4.

#### A.4 Proof of Theorem 1

We assume that 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.

We provide the detailed proof on the security of UPPRESSO. As analyzed above, the security requirements of UPPRESSO are that the system must ensure only the legitimate user can log into an honest RP under her unique account. We consider the visits to RP's resource paths are controlled by the visitors' cookies, so that the attacker can break the security only when he owns the cookie bound to the honest user. Therefore, we can propose the Definition 1 about the secure UPPRESSO system.

**Definition 1.** Let  $\mathcal{UWS}$  be a UPPRESSO web system,  $\mathcal{UWS}$  is secure **iff** for any honest RP  $r \in \mathcal{W}$  and the authenticated cookie  $c$  for honest  $u$ ,  $c$  is unknown to the attacker  $a$  and only  $c$  is used by  $u$ .

Therefore, the proof of Theorem 1 is converted into whether the UPPRESSO system meets the requirement in Definition 1. However, as we consider the attacker initially does not know any honest user's cookie, the requirement of Definition 1 can

be separated as the following requirements. Before describing the requirements, we firstly define the user  $u$ 's authenticated cookie for RP  $r$  as  $c(u, r)$ .

**Requirement 1.** If  $c(u, r)$  is the authenticated cookie owned by  $u$ ,  $c(u, r)$  cannot be obtained by  $a$ .

**Requirement 2.** If  $c$  is an unauthenticated cookie owned by  $a$ ,  $c$  cannot be set as  $c(u, r)$ .

**Requirement 3.** The user  $u$  does not use the attacker's cookie (denoted as  $c(a, r)$ ).

To prove that UPPRESSO meets the requirements, we now provide the following lemmas. Lemma 1 proves that the UPPRESSO system meets Requirement 1.

**Lemma 1.** Attacker does not learn users' cookies.

*Proof.* The Brute-force attacks, such as exhausting the possible users' cookies, are infeasible due to the length of cookies. The attackers can only try to obtain the cookies from honest processes in the system. For an honest user  $u$  and the honest RP  $r$ , the valid cookie  $c(u, r)$  can only be obtained by  $u$ 's browser  $b_u$ , the  $r$ 's script *script\_rp* and RP's server  $P^r$ . Here, we only need to prove that the attacker cannot receive the event from these processes which carries  $c(u, r)$ .

- $b_u$ . The browsers are considered honest and well implemented. Therefore, based on the same-origin policy,  $b_u$  only sends  $r$ 's cookie to RP's domain, so that attackers cannot receive the cookie.
- *script\_rp*. According to Algorithm 4, the *script\_rp* does not send any cookie.
- $P^r$ . According to Algorithm 2, the  $P^r$  does not send any cookie.

Therefore, Lemma 1 is proved.  $\square$

To prove UPPRESSO system meets Requirement 2, we need to know how the cookie can be set as  $c(u, r)$ . Based on Algorithm 2, we propose the following definition.

**Definition 2.** In  $\mathcal{UWS}$ , the cookie  $c$  is to be set as  $c(u, r)$  only when RP  $r$  receives a valid identity proof (denoted as  $t(u, r)$  here) of  $u$ , from the owner of  $c$ .

To prove that  $t(u, r)$  cannot be obtained by attackers, we introduce the following lemmas.

**Lemma 2.** Attacker does not learn users' passwords.

*Proof.* Same as the proof to Lemma 1, we only need to prove that attackers cannot receive the message from honest processes that carries the password. The honest IdP server is defined as  $P^i$  and the IdP script is defined as *script\_idp*. Here we give the proof about each processes.

- *script\_rp*. According to Algorithm 4, we can prove that RP script does not send any stored passwords.
- $P^r$ . According to Algorithm 2, it is easy to find out that RP server does not receive or send any stored passwords.
- *script\_idp*. Based on Algorithm 3, we can find that IdP script sends the user's password at Line 68. The target of this message is *Url* whose host is *IdPDomain* set at Line 67. The *IdPDomain* is set at Line 4 and the value is defined by the script and never modified. Therefore, the password can only be sent to the IdP server. The IdP server obtains the password at Line 10 in Algorithm 1 and does not send this parameter to any other processes.
- $P^i$ . Based on Algorithm 1, we can find that IdP server does not send any stored passwords.

Therefore, no attackers can obtain the password from honest processes, so that this lemma is proved.  $\square$

**Lemma 3.** Attacker cannot forge or modify the IdP-issued proofs.

*Proof.* The IdP-issued proofs include the *Cert* used in *script\_idp*, the *RegistrationResult* and *Token* used in  $P^r$ . 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 Algorithm 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 Algorithm 3. Therefore, *Cert* cannot be forged or modified.
- *RegistrationResult* is used in Algorithm 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 Algorithm 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.  $\square$

Here we now show the lemma to prove that UPPRESSO meets the requirements in Definition 2.

**Lemma 4.** Attacker cannot learn users' valid identity proofs.

*Proof.* As the *Token* has been proved that it can not be forged by the attackers, here we only need to prove that attackers cannot receive *Token* from other honest processes.

- Attacker cannot obtain the *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 Algorithm 2. It is easy to prove that the RP server does not send any *Token* to other processes.
- Attacker cannot obtain the *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 Algorithm 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 *Token* from IdP server. Considering the messages at Line 4, 12, 16, 23, 26, 36, 44, 51, 67 in Algorithm 1, we find that only the message at Line 67 carries the *Token*. This *Token* is generated at Line 65, following the trace where the *Content* at Line 63, the *PID<sub>U</sub>* at Line 61, the *ID<sub>U</sub>* at Line 60, the *session* at Line 48, and finally the *cookie* at Line 47. That is, the receiver of *Token* must be the owner of the *cookie* in which session that saves the parameter *ID<sub>U</sub>*. The *ID<sub>U</sub>* is set at Line 15 after verifying the password and never modified. As we have already proved that the cookies and passwords cannot be known to attackers, attackers cannot obtain the *Token* from IdP server.
- Attacker cannot obtain the *Token* from IdP script. As the proof provided above, only IdP sends the *Token* with the message at Line 67 in Algorithm 1, the IdP script can only receive the *Token* at Line 99 in Algorithm 3. Here we are going to prove that the token  $t(u, r)$  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<sub>RP</sub>* in  $t(u, r)$  as  $p$ . That is the *PID<sub>RP</sub>* at Line 69 in Algorithm 2 must equal to  $p$  and the *PID<sub>RP</sub>* 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 registration result. As the IdP checks the uniqueness of *PID<sub>RP</sub>* at Line 32 in Algorithm 1. The  $r$  and IdP script must share the same *RegistrationToken*. As the *RegistrationToken* contains the  $\text{Hash}(N_U)$ , the



IdP script and  $r$  must share the same  $ID_{RP}$ . Therefore, the  $Cert$  saved as the IdP scriptstate parameter must belong to  $r$ .

Therefore, attackers cannot learn users' valid identity proofs.  $\square$

So far we have proved that UPPRESSO meets the requirements in Definition 2. Requirement 2 is satisfied.

Then, we prove that UPPRESSO meets Requirement 3. As the browser follows the same-origin policy, the attackers cannot set its cookie to user's browser at RP's origin. Therefore, due to Definition 2, the user only sets her cookie as  $c(a, r)$  when RP receives the *Token* containing the attacker's  $PID_U$  and a valid  $PID_{RP}$  negotiated by  $u$  and  $r$ . It requires that the attackers must know a valid  $PID_{RP}$ . Here we give a lemma.

**Lemma 5.** Attacker does not know a valid  $PID_{RP}$  negotiated by user  $u$  and RP  $r$ .

*Proof.* Here we give the proof that attacker cannot obtain the  $PID_{RP}$  and  $N_U$  from each processes.

- $P^i$ . We can find in Algorithm 1, IdP only returns the message containing  $PID_{RP}$  to other processes when the  $PID_{RP}$  is included in the request message.
- $P^r$ . Same as IdP server, RP server only sends the message containing  $PID_{RP}$  at Line 55 in Algorithm 2, and the  $PID_{RP}$  is contained in the *RegistrationResult* received at Line 28 and verified at Line 44.
- *script\_rp*. We can find in Algorithm 4, RP script only sends the messages to RP server and IdP script. The receivers' identities are ensured at Line 3, 4. Therefore, attackers cannot obtains the  $PID_{RP}$  and  $N_U$ .
- *script\_idp*. The HTTP requests sent by IdP script are forwarded to the domain set at Line 4 in Algorithm 3. The HTTP requests are sent to IdP server. The postMessages are sent to the one set at Line 3, and we will prove the target cannot be attacker. According to Line 44 in Algorithm 2,  $PID_{RP}$  is valid at RP server only when RP server receives the registration result. The Hash( $N_U$ ) in the result ensures the result must be issued for the correct  $ID_{RP}$ . As the registration result is  $PID_{RP}$ -unique due to Line 32 in Algorithm 1, the registration result received by IdP script at Line 35 in Algorithm 3 must be same as the one in RP server. This HTTP response is related with the HTTP request at Line 28, carrying  $PID_{RP}$  and Hash( $N_U$ ) at Line 21, 25. It ensures the  $Cert$  obtained at Line 15 must belong to RP. As the target at Line 3 is the window which opens the IdP script window and asks for user's login consent, user can easily find out the target site is not coincident with the consent requirement. Therefore, the target cannot be the attacker.  $\square$

Therefore, Requirement 3 is satisfied and Theorem 1 is proved.

## B Algorithm

### B.1 IdP process

---

#### Algorithm 1 $R^i$

---

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

```
1: let  $s := s'$ 
2: let  $n, method, path, parameters, headers, body$  such that
    $\langle \text{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$ 
   if possible; otherwise stop  $\langle \rangle, s'$ 
3: if  $path \equiv /script$  then
4:   let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, \text{IdPScript} \rangle$ 
5:   stop  $\langle f, a, m' \rangle, s'$ 
6: else if  $path \equiv /login$  then
7:   let  $cookie := headers[Cookie]$ 
8:   let  $session := s'.sessions[cookie]$ 
9:   let  $username := body[username]$ 
10:  let  $password := body[password]$ 
11:  if  $password \neq \text{SecretOfID}(username)$  then
12:    let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, \text{LoginFailure} \rangle$ 
13:    stop  $\langle f, a, m' \rangle, s'$ 
14:  end if
15:  let  $session[uid] := \text{UIDOfUser}(username)$ 
16:  let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, \text{LoginSuccess} \rangle$ 
17:  stop  $\langle f, a, m' \rangle, s'$ 
18: else if  $path \equiv /loginInfo$  then
19:   let  $cookie := headers[Cookie]$ 
20:   let  $session := s'.sessions[cookie]$ 
21:   let  $username := session[username]$ 
22:   if  $username \neq \text{null}$  then
23:     let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, \text{Logged} \rangle$ 
24:     stop  $\langle f, a, m' \rangle, s'$ 
25:   end if
26:   let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, \text{Unlogged} \rangle$ 
27:   stop  $\langle f, a, m' \rangle, s'$ 
28: else if  $path \equiv /dynamicRegistration$  then
29:   let  $PID_{RP} := body[PID_{RP}]$ 
30:   let  $Endpoint := body[Endpoint]$ 
31:   let  $Nonce := body[Nonce]$ 
32:   if  $PID_{RP} \in \text{ListOfPID}()$  then
33:     let  $Content := \langle \text{Fail}, PID_{RP}, Nonce \rangle$ 
34:     let  $Sig := \text{Sig}(Content, s'.SignKey)$ 
35:     let  $RegistrationResult := \langle Content, Sig \rangle$ 
36:     let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, RegistrationResult \rangle$ 
37:     stop  $\langle f, a, m' \rangle, s'$ 
38:   end if
39:   let  $Validity := \text{CurrentTime}() + s'.Validity$ 
40:   let  $s'.RPs := s'.RPs + \langle \rangle \langle PID_{RP}, Endpoint, Validity \rangle$ 
41:   let  $Content := \langle \text{OK}, PID_{RP}, Nonce, Validity \rangle$ 
42:   let  $Sig := \text{Sig}(Content, s'.SignKey)$ 
43:   let  $RegistrationResult := \langle Content, Sig \rangle$ 
44:   let  $m' := \langle \text{HTTPResp}, n, 200, \langle \rangle, RegistrationResult \rangle$ 
45:   stop  $\langle f, a, m' \rangle, s'$ 
46: else if  $path \equiv /authorize$  then
```

---

```

47: let cookie := headers[Cookie]
48: let session := s'.sessions[cookie]
49: let username := session[username]
50: if username ≡ null then
51:   let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
52:   stop ⟨f, a, m'⟩, s'
53: end if
54: let PIDRP := parameters[PIDRP]
55: let Endpoint := parameters[Endpoint]
56: if PIDRP ∉ ListOfPID() ∨ Endpoint ∉ EndpointsOfRP(PIDRP) then
57:   let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
58:   stop ⟨f, a, m'⟩, s'
59: end if
60: let IDU := session[uid]
61: let PIDU := Multiply(PIDRP, IDU)
62: let Validity := CurrentTime() + s'.Validity
63: let Content := ⟨PIDRP, PIDU, s'.ID, Validity⟩
64: let Sig := Sig(Content, s'.SignKey)
65: let Token := ⟨Content, Sig⟩
66: let s'.Tokens := s'.Tokens + ⟨⟩ Token
67: let m' := ⟨HTTPResp, n, 200, ⟨⟩, ⟨Token, Token⟩⟩
68: stop ⟨f, a, m'⟩, s'
69: end if
70: stop ⟨⟩, s'

```

---

## B.2 RP process

---

### Algorithm 2 $R^r$

---

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

```

1: let s := s'
2: let n, method, path, parameters, headers, body such that
   ⟨HTTPReq, n, method, path, parameters, headers, body⟩ ≡ m
   if possible; otherwise stop ⟨⟩, s'
3: if path ≡ /script then
4:   let m' := ⟨HTTPResp, n, 200, ⟨⟩, RPScript⟩
5:   stop ⟨f, a, m'⟩, s'
6: else if path ≡ /login then
7:   let m' := ⟨HTTPResp, n, 302, ⟨⟨Location, s'.IdP.ScriptUrl⟩⟩, ⟨⟩⟩
8:   stop ⟨f, a, m'⟩, s'
9: else if path ≡ /startNegotiation then
10:  let cookie := headers[Cookie]
11:  let session := s'.sessions[cookie]
12:  let NU := parameters[NU]
13:  let PIDRP := Multiply(s'.IDRP, NU)
14:  let T := ExEU(NU, s'.IdP.q)
15:  let session[NU] := NU
16:  let session[PIDRP] := PIDRP
17:  let session[t] := T
18:  let session[state] := expectRegistration
19:  let m' := ⟨HTTPResp, n, 200, ⟨⟩, ⟨Cert, s'.Cert⟩⟩
20:  stop ⟨f, a, m'⟩, s'
21: else if path ≡ /registrationResult then
22:  let cookie := headers[Cookie]
23:  let session := s'.sessions[cookie]

```

```

24: if  $session[state] \neq expectRegistration$  then
25:   let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
26:   stop  $\langle f, a, m' \rangle, s'$ 
27: end if
28: let  $RegistrationResult := body[RegistrationResult]$ 
29: let  $Content := RegistrationResult.Content$ 
30: if  $checksig(Content, RegistrationResult.Sig, s'.IdP.PubKey) \equiv FALSE$  then
31:   let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
32:   let  $session := null$ 
33:   stop  $\langle f, a, m' \rangle, s'$ 
34: end if
35: if  $Content.Result \neq OK$  then
36:   let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
37:   let  $session := null$ 
38:   stop  $\langle f, a, m' \rangle, s'$ 
39: end if
40: let  $PID_{RP} := session[PID_{RP}]$ 
41: let  $N_U := session[N_U]$ 
42: let  $Nonce := Hash(N_U)$ 
43: let  $Time := CurrentTime()$ 
44: if  $PID_{RP} \neq Content.PID_{RP} \vee Nonce \neq Content.Nonce \vee Time > Content.Validity$  then
45:   let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
46:   let  $session := null$ 
47:   stop  $\langle f, a, m' \rangle, s'$ 
48: end if
49: let  $session[PIDValidity] := Content.Validity$ 
50: let  $Endpoint \in s'.Endpoints$ 
51: let  $session[state] := expectToken$ 
52: let  $Nonce' := Random()$ 
53: let  $session[Nonce] := Nonce'$ 
54: let  $Body := \langle PID_{RP}, Endpoint, Nonce' \rangle$ 
55: let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Body \rangle$ 
56: stop  $\langle f, a, m' \rangle, s'$ 
57: else if  $path \equiv /uploadToken$  then
58:   let  $cookie := headers[Cookie]$ 
59:   let  $session := s'.sessions[cookie]$ 
60:   if  $session[state] \neq expectToken$  then
61:     let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
62:     stop  $\langle f, a, m' \rangle, s'$ 
63:   end if
64:   let  $Token := body[Token]$ 
65:   if  $checksig(Token.Content, Token.Sig, s'.IdP.PubKey) \equiv FALSE$  then
66:     let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
67:     stop  $\langle f, a, m' \rangle, s'$ 
68:   end if
69:   let  $PID_{RP} := session[PID_{RP}]$ 
70:   let  $Time := CurrentTime()$ 
71:   let  $PIDValidity := session[PIDValidity]$ 
72:   let  $Content := Token.Content$ 
73:   if  $PID_{RP} \neq Content.PID_{RP} \vee Time > Content.Validity \vee Time > PIDValidity$  then
74:     let  $m' := \langle HTTPResp, n, 200, \langle \rangle, Fail \rangle$ 
75:     stop  $\langle f, a, m' \rangle, s'$ 
76:   end if
77:   let  $PID_U := Content.PID_U$ 

```



```

78: let  $T := session[t]$ 
79: let  $Account := Multiply(PID_U, T)$ 
80: if  $Account \in ListOfUser()$  then
81:   let RegisterUser( $Account$ )
82: end if
83: let  $session[user] := Account$ 
84: let  $m' := \langle HTTPResp, n, 200, \langle \rangle, LoginSuccess \rangle$ 
85: stop  $\langle f, a, m' \rangle, s'$ 
86: end if
87: stop  $\langle \rangle, s'$ 

```

---

### B.3 IdP scripting process

---

#### Algorithm 3 *script\_idp*

---

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

```

1: let  $s' := scriptstate$ 
2: let  $command := \langle \rangle$ 
3: let  $target := PARENTWINDOW(tree, docnonce)$ 
4: let  $IdPDomain := s'.IdPDomain$ 
5: switch  $s'.q$  do
6:   case start:
7:     let  $N_U := Random()$ 
8:     let  $command := \langle POSTMESSAGE, target, \langle \langle N_U, N_U \rangle \rangle, null \rangle$ 
9:     let  $s'.Parameters[N_U] := N_U$ 
10:    let  $s'.q := expectCert$ 
11:   case expectCert:
12:     let  $pattern := \langle POSTMESSAGE, *, Content, * \rangle$ 
13:     let  $input := CHOOSEINPUT(scriptinputs, pattern)$ 
14:     if  $input \neq null$  then
15:       let  $Cert := input.Content[Cert]$ 
16:       if  $checksig(Cert.Content, Cert.Sig, s'.PubKey) \equiv null$  then
17:         let stop  $\langle \rangle$ 
18:       end if
19:       let  $s'.Parameters[Cert] := Cert$ 
20:       let  $N_U := s'.Parameters[N_U]$ 
21:       let  $PID_{RP} := Multiply(Cert.Content.ID_{RP}, N_U)$ 
22:       let  $s'.Parameters[PID_{RP}] := PID_{RP}$ 
23:       let  $Endpoint := RandomUrl()$ 
24:       let  $s'.Parameters[Endpoint] := Endpoint$ 
25:       let  $Nonce := Hash(N_U)$ 
26:       let  $Url := \langle URL, S, IdPDomain, /dynamicRegistration, \langle \rangle \rangle$ 
27:       let  $s'.refXHR := Random()$ 
28:       let  $command := \langle XMLHTTPREQUEST, Url, POST, \langle \langle PID_{RP}, PID_{RP} \rangle, \langle Nonce, Nonce \rangle, \langle Endpoint, Endpoint \rangle \rangle, s'.refXHR \rangle$ 
29:       let  $s'.q := expectRegistrationResult$ 
30:     end if
31:   case expectRegistrationResult:
32:     let  $pattern := \langle XMLHTTPREQUEST, Body, s'.refXHR \rangle$ 
33:     let  $input := CHOOSEINPUT(scriptinputs, pattern)$ 
34:     if  $input \neq null \wedge input.Content[RegistrationResult].type \equiv OK$  then
35:       let  $RegistrationResult := input.Body[RegistrationResult]$ 
36:       if  $RegistrationResult.Content.Result \neq OK$  then
37:         let  $s'.q := stop$ 
38:       end if

```

```

39:     end if
40:     let command := ⟨POSTMESSAGE, target, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, null⟩
41:     let s'.q := expectProofRequest
42: end if
43: case expectProofRequest:
44:   let pattern := ⟨POSTMESSAGE, *, Content, *⟩
45:   let input := CHOOSEINPUT(scriptinputs, pattern)
46:   if input ≠ null then
47:     let PIDRP := input.Content[PIDRP]
48:     let EndpointRP := input.Content[Endpoint]
49:     let s'.Parameters[Nonce] := input.Content[Nonce]
50:     let Cert := s'.Parameters[Cert]
51:     if EndpointRP ∉ Cert.Content.Endpoints ∨ PIDRP ≠ s'.Parameters[PIDRP] then
52:       let s'.q := stop
53:       let stop ⟨⟩
54:     end if
55:     let s'.Parameters[EndpointRP] := EndpointRP
56:     let Url := ⟨URL, S, IdPDomain, /loginInfo, ⟨⟩⟩
57:     let s'.refXHR := Random()
58:     let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
59:     let s'.q := 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 username ∈ ids
67:       let Url := ⟨URL, S, IdPDomain, /login, ⟨⟩⟩ mystates'.refXHR := Random()
68:       let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨username, username⟩, ⟨password, secret⟩⟩, s'.refXHR⟩
69:       let s'.q := expectLoginResult
70:     else if input.Body ≡ Unlogged then
71:       let PIDRP := s'.Parameters[PIDRP]
72:       let Endpoint := s'.Parameters[Endpoint]
73:       let Nonce := s'.Parameters[Nonce]
74:       let Url := ⟨URL, S, IdPDomain, /authorize,
75:         ⟨⟨PIDRP, PIDRP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩
76:       let s'.refXHR := Random()
77:       let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
78:       let s'.q := expectToken
79:     end if
80:   end if
81: case expectLoginResult:
82:   let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
83:   let input := CHOOSEINPUT(scriptinputs, pattern)
84:   if input ≠ null then
85:     if input.Body ≠ LoginSuccess then
86:       let stop ⟨⟩
87:     end if
88:     let PIDRP := s'.Parameters[PIDRP]
89:     let Endpoint := s'.Parameters[Endpoint]
90:     let Nonce := s'.Parameters[Nonce]
91:     let Url := ⟨URL, S, IdPDomain, /authorize,
92:       ⟨⟨PIDRP, PIDRP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩

```

```

91:   let  $s'.refXHR := \text{Random}()$ 
92:   let  $command := \langle \text{XMLHTTPREQUEST}, Url, \text{GET}, \langle \rangle, s'.refXHR \rangle$ 
93:   let  $s'.q := expectToken$ 
94: end if
95: case expectToken:
96:   let  $pattern := \langle \text{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$ 
97:   let  $input := \text{CHOOSEINPUT}(scriptinputs, pattern)$ 
98:   if  $input \neq \text{null}$  then
99:     let  $Token := input.Body[Token]$ 
100:    let  $RPOrigin := \langle s'.Parameters[Endpoint_{RP}], S \rangle$ 
101:    let  $command := \langle \text{POSTMESSAGE}, target, \langle Token, Token \rangle, RPOrigin \rangle$ 
102:    let  $s.q := stop$ 
103:   end if
104: end switch
105: let stop  $\langle s', cookies, localStorage, sessionStorage, command \rangle$ 

```

---

#### B.4 RP scripting process

---

##### Algorithm 4 *script\_rp*

---

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

```

1: let  $s' := scriptstate$ 
2: let  $command := \langle \rangle$ 
3: let  $IdPWindow := \text{SUBWINDOW}(tree, docnonce).nonce$ 
4: let  $RPDomain := s'.RPDomain$ 
5: let  $IdPOrigin := \langle s'.IdPDomain, S \rangle$ 
6: switch  $s'.q$  do
7:   case start:
8:     let  $Url := \langle \text{URL}, S, RPDomain, /login, \langle \rangle \rangle$ 
9:     let  $command := \langle \text{IFRAME}, Url, _SELF \rangle$ 
10:    let  $s'.q := expectN_U$ 
11:   case expectN_U:
12:     let  $pattern := \langle \text{POSTMESSAGE}, *, Content, * \rangle$ 
13:     let  $input := \text{CHOOSEINPUT}(scriptinputs, pattern)$ 
14:     if  $input \neq \text{null}$  then
15:       let  $N_U := input.Content[N_U]$ 
16:       let  $Url := \langle \text{URL}, S, RPDomain, /startNegotiation, \langle \rangle \rangle$ 
17:       let  $s'.refXHR := \text{Random}()$ 
18:       let  $command := \langle \text{XMLHTTPREQUEST}, Url, \text{POST}, \langle \langle N_U, N_U \rangle \rangle, s'.refXHR \rangle$ 
19:       let  $s'.q := expectCert$ 
20:     end if
21:   case expectCert:
22:     let  $pattern := \langle \text{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$ 
23:     let  $input := \text{CHOOSEINPUT}(scriptinputs, pattern)$ 
24:     if  $input \neq \text{null}$  then
25:       let  $Cert := input.Content[Cert]$ 
26:       let  $command := \langle \text{POSTMESSAGE}, IdPWindow, \langle \langle Cert, Cert \rangle \rangle, IdPOrigin \rangle$ 
27:       let  $s'.q := expectRegistrationResult$ 
28:     end if
29:   case expectRegistrationResult:
30:     let  $pattern := \langle \text{POSTMESSAGE}, *, Content, * \rangle$ 
31:     let  $input := \text{CHOOSEINPUT}(scriptinputs, pattern)$ 
32:     if  $input \neq \text{null}$  then
33:       let  $RegistrationResult := input.Content[RegistrationResult]$ 
34:       let  $Url := \langle \text{URL}, S, RPDomain, /registrationResult, \langle \rangle \rangle$ 

```

```

35:     let  $s'.refXHR := \text{Random}()$ 
36:     let  $command := \langle \text{XMLHTTPREQUEST}, Url, \text{POST}, \langle \langle \text{RegistrationResult}, \text{RegistrationResult} \rangle \rangle, s'.refXHR \rangle$ 
37:     let  $s'.q := \text{expectTokenRequest}$ 
38:   end if
39: case  $\text{expectTokenRequest}$ :
40:   let  $pattern := \langle \text{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$ 
41:   let  $input := \text{CHOOSEINPUT}(\text{scriptinputs}, pattern)$ 
42:   if  $input \neq \text{null}$  then
43:     let  $PID_{RP} := input.Content.Body[PID_{RP}]$ 
44:     let  $Endpoint := input.Content.Body[Endpoint]$ 
45:     let  $Nonce := input.Content.Body[Nonce]$ 
46:     let  $command := \langle \text{POSTMESSAGE}, IdPWindow,$ 
47:        $\langle \langle PID_{RP}, PID_{RP} \rangle, \langle Endpoint, Endpoint \rangle, \langle Nonce, Nonce \rangle \rangle, IdPOrigin \rangle$ 
48:     let  $s'.q := \text{expectToken}$ 
49:   end if
50: case  $\text{expectToken}$ :
51:   let  $pattern := \langle \text{POSTMESSAGE}, *, Content, * \rangle$ 
52:   let  $input := \text{CHOOSEINPUT}(\text{scriptinputs}, pattern)$ 
53:   if  $input \neq \text{null}$  then
54:     let  $Token := input.Content[Token]$ 
55:     let  $Url := \langle \text{URL}, S, RPDomain, /uploadToken, \rangle \rangle$ 
56:     let  $s'.refXHR := \text{Random}()$ 
57:     let  $command := \langle \text{XMLHTTPREQUEST}, Url, \text{POST}, \langle \langle Token, Token \rangle \rangle, s'.refXHR \rangle$ 
58:     let  $s'.q := \text{expectLoginResult}$ 
59:   end if
60: case  $\text{expectLoginResult}$ :
61:   let  $pattern := \langle \text{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$ 
62:   let  $input := \text{CHOOSEINPUT}(\text{scriptinputs}, pattern)$ 
63:   if  $input \neq \text{null}$  then
64:     if  $input.Body \equiv \text{LoginSuccess}$  then
65:       let  $\text{LoadHomepage}$ 
66:     end if
67:   end if
68: end switch

```

---