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) [**?**], OAuth [**?**] and SAML [**?**] have been widely deployed in today's Internet for identity management and authentication. 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Besides, the adoption of SSO raises public concern about user privacy [**?**, **?**, **?**, **?**], 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. **??**, 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 [**?**, **?**]. Similarly, by design, the RPs can learn users' identities from the received identify proofs. If the IdP uses a unique or related user identifier(s) to generate identity proofs for her to access different RPs [**?**, **?**], curious RPs can collude to link the login requests, associate her attributes across multiple RPs, and track her online activities [**?**]. 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

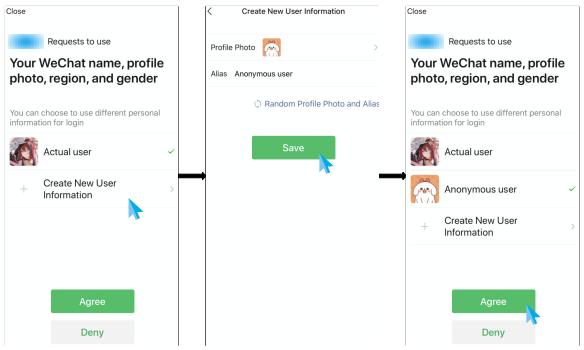

Figure 1: Anonymous SSO login in WeChat.

more prominent and severe than it was in the past. Recently, many IdPs have taken user privacy in SSO into serious consideration and offered new protection against RP-based identity linkage. For example, Figure **??** 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 [**?**, **?**], and Active Directory Federation Services [**?**] and Oracle Access Management [**?**] 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 [**?**] and Onavo [**?**]). 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 [**?**] and private set intersection [**?**] 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 [**?**, **?**, **?**, **?**]. 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 **??**). 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 problem* 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 **??**). 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 [**?**] 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 [**?**] and SPRESSO [**?**]) and demonstrate its efficiency.

The rest of the paper is organized as follows. We introduce the background in Section **??**, the privacy problem and our design ideas in Section **??**, and the threat model in Section **??**. Section **??** describes the detailed design of UPPRESSO based on identifier transformation, followed by a formal analysis of the privacy and security in Sections **??**

and **??**. We explain the implementation specifics and experimental evaluation in Section **??**, discuss the extensions and related works in Sections **??** and **??**, and conclude our work in Section **??**.

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 [**?**]. 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 **??**.

As shown in Figure **??**, 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* [**?**]. 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_{RP} registration* process (see details in Section **??**.C), which allows an RP to generate different privacy-preserving RP identifiers and register them with the IdP.

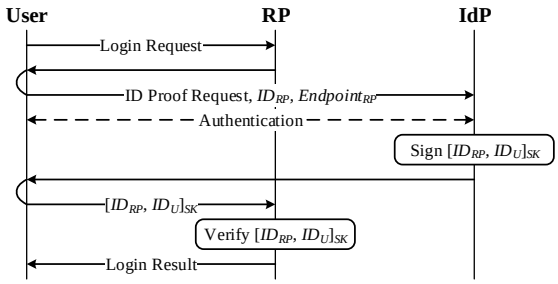

Figure 2: The implicit flow of OIDC.

2.2 Existing Privacy Solutions for SSO

Pairwise pseudonymous identifier (PPID) is recommended by NIST [**?**] and specified in several SSO protocols [**?**, **?**] 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 [**?**] and SPRESSO [**?**] 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 [**?**] and Firefox Accounts [**?**]), 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 **??** for reference.

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

3.1 Security Properties of SSO

The primary goal of SSO services is to support secure user authentication [**?**], 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 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 [**?**, **?**, **?**] and practical attacks [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

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) [**?**]. 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) [**?**, **?**, **?**, **?**, **?**], 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) [**?**], 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_{RP}s 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_Us 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 $Account$s 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 **??**. 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 $Account$s 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},

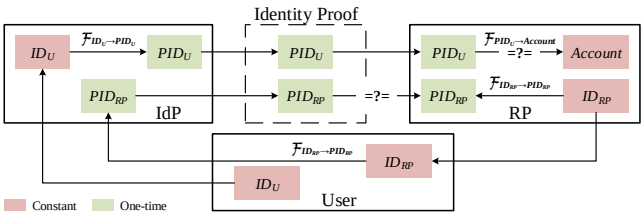

Figure 3: Identifier transformations in privacy-preserving SSO.

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_Us and PID_{RP}s are dynamically generated in different logins; (b) in each login session, PID_{RP} is used to 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 $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}(ID_{RP}, T)$ 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 $\mathcal{F}_{ID_U\mapsto PID_U}(ID_U, PID_{RP})$ for the IdP to generate PID_U. When an RP receives an identity proof, another transformation function $\mathcal{F}_{PID_U\mapsto Account}(PID_U, PID_{RP}, T)$ 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 [**?**], BrowserID [**?**] and SPRESSO [**?**]) to the identifier transformation framework in Figure **??** and summarize their potential privacy issues in Table **??**. 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_Us for a user at different RPs. There are deterministic one-to-many mappings from ID_U to PID_Us. However, the RP sends its real identifier to IdP in each login and thus suffers from IdP-based login tracing.

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†	ID_U	$\perp$	ID_U
UPPRESSO	$\mathcal{F}(ID_U, PID_{RP})$	$\mathcal{F}(ID_{RP}, T)$	$\mathcal{F}(PID_U, T)$

†: 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.

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 $\mathcal{F}_{ID_U\mapsto PID_U}$, $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$ and $\mathcal{F}_{PID_U\mapsto Account}$ 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 [**?**, **?**] 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, $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$, $\mathcal{F}_{ID_U\mapsto PID_U}$ and $\mathcal{F}_{PID_U\mapsto Account}$, 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:

$$\mathcal{F}_{ID_{RP}\mapsto PID_{RP}} : PID_{RP} = N_U \cdot ID_{RP} \tag{1}$$

With $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$, 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_{RP}s 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:

$$\mathcal{F}_{ID_U\mapsto PID_U} : PID_U = ID_U \cdot PID_{RP} \tag{2}$$

From Equations **??** and **??**, 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:

$$\mathcal{F}_{PID_U\mapsto Account} : A = T \cdot PID_U \tag{3}$$

From Equations **??**, **??** and **??**, 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_Us 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, UPRESSO supports two desirable properties discussed in Section **??** 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 UPRESSO, 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 **??**.

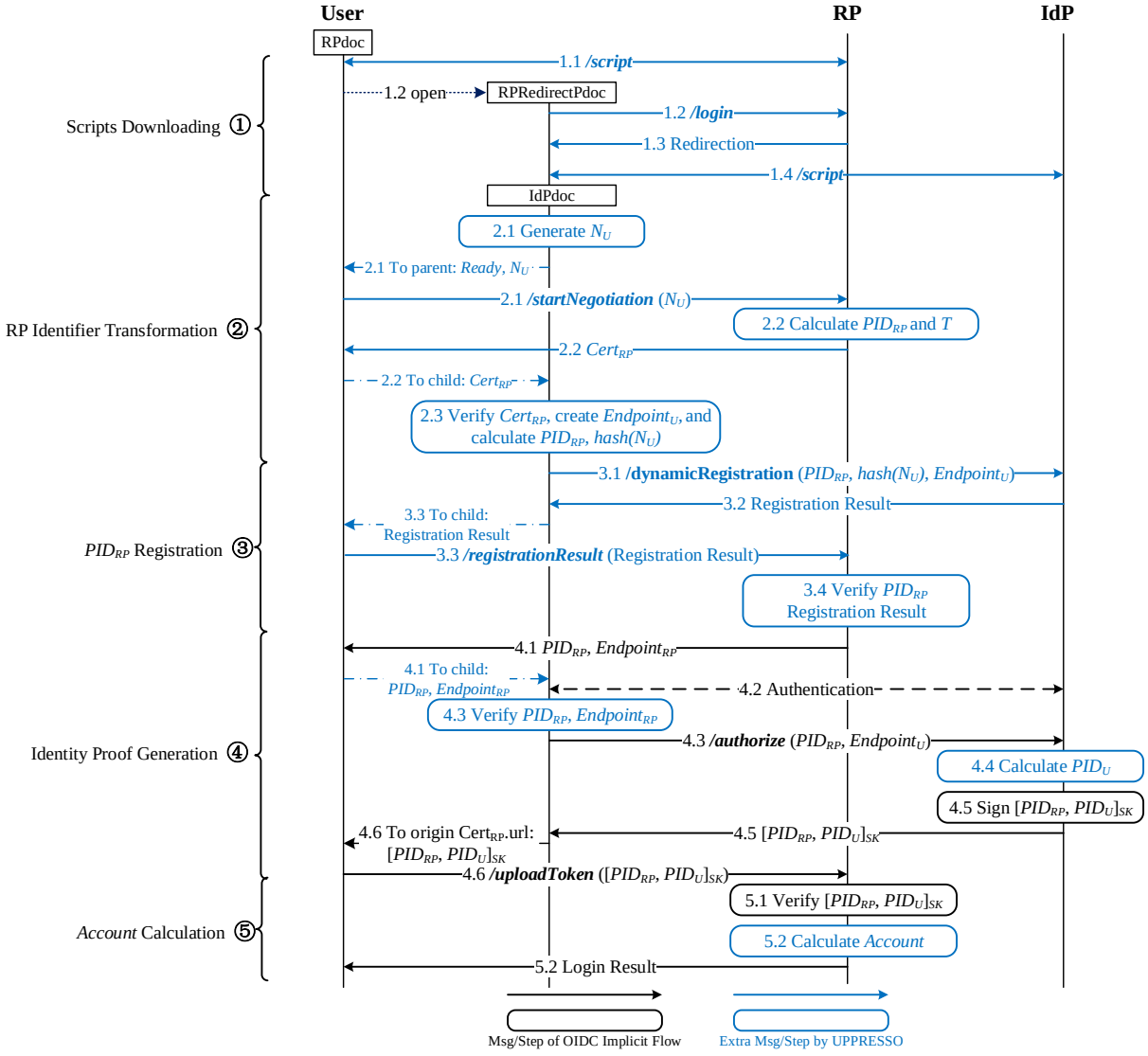

Figure 4: The flow of a user login in UPPRESSO.

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 $N_U \neq 0 \bmod n$, calculates PID_{RP} and derives the trapdoor $T = {N_U}^{-1} \bmod n$. 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. **$\mathbf{PID_{RP}}$** *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_{RP}s 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_{RP}s and N_Us. 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 $Accout = T \cdot PID_U$, 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 [**?**], 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 **??**).

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 **??**, the only information that is related to the RP's identity and is 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 ha no control of the process, the IdP

should treat PID_{RP} as being 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 [**?**]. Here, we briefly introduce the DDH assumption: Let q be a large prime and $\mathbb{G}$ denotes 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 $\sigma(n)$ with the security parameter n such that:

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

RP-based identity linkage. Collusive RPs can act arbitrarily to correlate PID_Us 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 **??**). RP-based identity linkage is impossible in UPPRESSO *if and only if the adversary has no advantage over the challenger in the guessing game.*

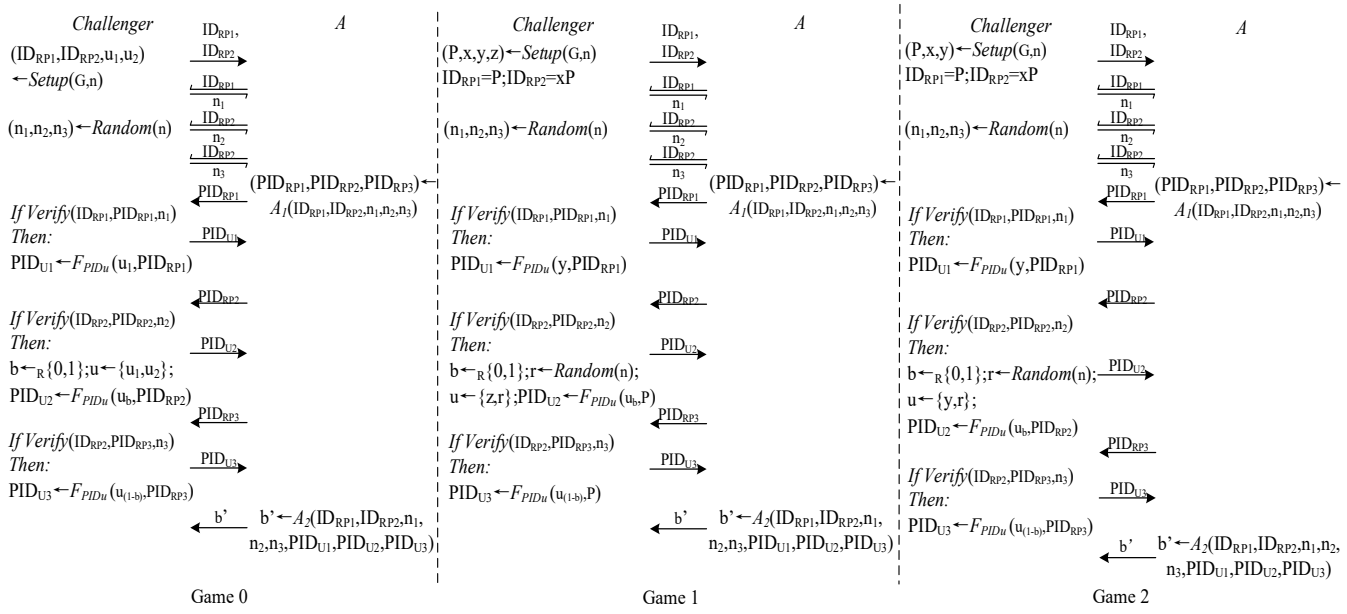

Figure 5: Interactions between the challenger and the adversary in three Games.

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

- *Initialization:* In initialization, the challenger generates ID_{RP}s and ID_Us for multiple RPs and users using the initialization algorithm $Setup(G,n)$, where G and n are defined in table **??**.

- *Random number generation:* When the challenger receives the ID_{RP}s from the adversary, it generates a random $N_U \in \mathbb{Z}_n$ for each ID_{RP} using the algorithm $Random(n)$.

- *PID_U generation:* When the challenger receives an PID_{RP}, it first verifies if the PID_{RP} is generated for ID_{RP} with the corresponding N_U, using the algorithm $Verify(ID_{RP}, PID_{RP}, N_U)$. Then, it generates the PID_U with the algorithm $F_{PID_U}(ID_U, PID_{RP})$ and sends it to the adversary.

To prove the privacy of UPPRESSO against RP-based identity linkage, we define three Games, as shown in figure **??**. First, based on the above description, we model the ID_U-guessing game following the UPPRESSO design as `Game0` : (1) First, the adversary receives two ID_{RP}s (i.e., ID_{RP1} and ID_{RP2}) and three N_Us (i.e., n_1, n_2, and n_3). It then generates three PID_{RP}s accordingly. From the challenger's view, three PID_{RP}s are related to three N_Us, respectively. (2) Then, the challenger generates PID_Us for different PID_{RP}s, using two ID_Us (i.e., u_1 and u_2). In particular, the challenger generates PID_{U1} from ID_{U1} directly, and selects a random number $b \in \{0,1\}$ to generate $PID_{U2} = ID_{Ub} \cdot PID_{RP2}$, and $PID_{U3} = ID_{U(1-b)} \cdot PID_{RP3}$. (3) Finally, the adversary sends its guess b' to the challenger. If $b' = b$, the adversary wins the game.

We define the event $[b' = b]$ in `Game0` as Γ. If the adversary has no advantage in guessing b correctly, which indicates the PID_Us are generated from the same ID_U, the prob-

ability $Pr[\Gamma]$ should be 1/2. Therefore, we conclude that in Game0, UPPRESSO is secure against RP-based identity linkage if and only if $Pr[\Gamma] = 1/2$.

Next, we build the ideal model of the guessing game, denoted as Game1. In this model, the probability that the adversary correctly guessing b is 1/2. This time, the challenger randomly selects z and r and uses them to generate PID_{U2} and PID_{U3}, respectively. Since the adversary does not know z and r, he does not know b neither. Similarly, we define the event $[b' = b]$ in Game1 as Γ_1, and $Pr[\Gamma_1]$ should be 1/2.

According to the DDH assumption, we need to prove that $|Pr[\Gamma_1] - Pr[\Gamma]| = \sigma(n)$, where $\sigma(n)$ is negligible. So, we build another model of Game, denoted as Game2, by setting the values of the parameters defined in Game0 to be: $ID_{RP2} = xID_{RP1}$, $u_1 = y$ and $u_2 = r$, where r is a random number. Again, we define the event $[b' = b]$ in Game2 as Γ_2, and $Pr[\Gamma_2] = Pr[\Gamma]$ should be true.

After defining the three games, we prove that $|Pr[\Gamma_1] - Pr[\Gamma_2]| = \sigma(n)$ as follows. In each game, the adversary uses the algorithm A_2 to derive b' from the collected data, i.e., $\{ID_{RP1}, ID_{RP2}, n_1, n_2, n_3, PID_{U1}, PID_{U2}, PID_{U3}\}$. Now, let us replace the parameters in Game1 and Game2 with the exact values:

$$b'_{game1} \leftarrow A_2(P, xP, n_1, n_2, n_3, yn_1P, zP, rP)$$
$$b'_{game2} \leftarrow A_2(P, xP, n_1, n_2, n_3, yn_1P, xyn_2P, rn_3P)$$
$$or\ b'_{game2} \leftarrow A_2(P, xP, n_1, n_2, n_3, yn_1P, rn_2P, xyn_3P)$$

Since n_1, n_2, n_3 are randomly chosen by the challenger, which are not related to ID_U, the adversary can easily remove them and obtain $b'_{game1} \leftarrow A_2(P, xP, yP, zP, rP)$ in Game1 and $b'_{game2} \leftarrow A_2(P, xP, yP, xyP, xrP)$ in Game2.

As r is randomly chosen and unknown to the adversary, rP and xrP are also random points. We can rewrite b'_{game1} and b'_{game2} as $b'_{game1} \leftarrow A_2(P, xP, yP, zP, r_1P)$ and $b'_{game2} \leftarrow A_2(P, xP, yP, xyP, r_2P)$, which means there is no non-negligible difference between the success probability in Game1 and Game2, according to the DDH assumption. Otherwise, we should be able to build a PPT distinguishing algorithm that breaks DDH assumption about the adversary.

Such distinguishing algorithm D is shown in figure **??**. The inputs of the algorithm is $\{P, X, Y, Z\}$. To the adversary, it is Game1 if the input is in the form $\{P, xP, yP, zP\}_{x,y,z \in \mathbb{Z}_n}$, and it is Game1 if the input is $\{P, xP, yP, xyP\}_{x,y \in \mathbb{Z}_n}$. As a result,

$$Pr[D(P, xP, yP, zP) = 1] = Pr[\Gamma_1]$$
$$Pr[D(P, xP, yP, xyP) = 1] = Pr[\Gamma_2]$$

Therefore, $|Pr[\Gamma_1] - Pr[\Gamma_2]| = \sigma(n)$, where $\sigma(n)$ is negligible, and n is the security parameter. It means the adversary has no advantage in guessing b in Game0. Therefore, he cannot distinguish if two PID_Us at two different RPs belong to the same user or not. This proves that UPPRESSO is resistant to RP-based identity linkage attacks.

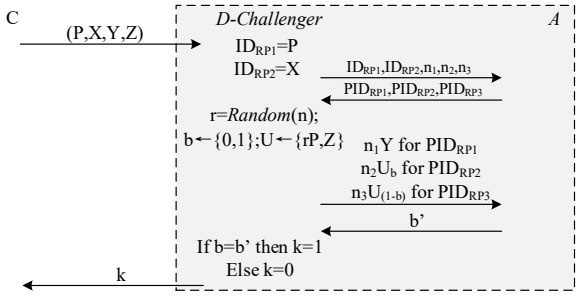

Figure 6: Distinguishing algorithm.

7 Security Analysis

We formally analyze the security properties of UPPRESSO based on the Dolev-Yao style web model [**?**], which has been widely used in the formal analysis of SSO protocols such as OAuth 2.0 [**?**] and OIDC [**?**]. 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 [**?**] 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*. [**?**] 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 [**?**].

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 the form $(\mathcal{W}, \mathcal{S}, \texttt{script}, E^0)$, 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, 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 $\mathcal{UWS} = (\mathcal{W},\mathcal{S},\texttt{script},E^0)$. $\mathcal{W}$ 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 script_rp, script_idp and script_attacker, where script_rp and script_idp are honest scripts downloaded by an RP process and the IdP process, and 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 $Account_1 = ID_{U_1}ID_{RP_1}$, $Account_2 = ID_{U_2}ID_{RP_2}$, and $Account_1 = Account_2$ 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) $ID_UPID_{RP_{adv}}$; or (c) $ID_{U_{adv}}PID_{RP}$. Due to the PID_{RP} registration, the (a) and (b) cannot be accepted by honest RP. In (c) the PID_{RP} must be created as $N_UID_{RP_{hon}}$, while the hash(N_U) is included in the registration result. Moreover, the trapdoor $T = N_U^{-1} \bmod n$ cannot be controlled by the 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 **??**. In this model, we consider only the network attackers 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 **??**, 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 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 **??** can be easily proved because the password is only sent by an honest IdP scripting process to the IdP server. Lemma **??** can be proved by showing that the proofs issued by the IdP process are signed and verified. With Lemma **??** and Lemma **??**, 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 **??**. According to the proof at 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_{RP}s with different users. According to this lemma, the adversary cannot

forge the token accepted by RP in an honest user's login session. Therefore, the requirement (C) is satisfied.

Finally, we prove $\mathcal{UWS}$ satisfies requirements (A) and (B) in Definition **??**. As a result, Theorem **??** 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 the 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 [**?**], an open-source OIDC Java implementation certificated by the OpenID Foundation [**?**]. 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 jsrsasign [**?**], an efficient JavaScript cryptographic library.

We provide a Java SDK for RPs to integrate UPPRESSO. The SDK provides two functions to encapsulate the RP's processing: 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 1,000 lines code and cryptographic computations are implemented based on the 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.

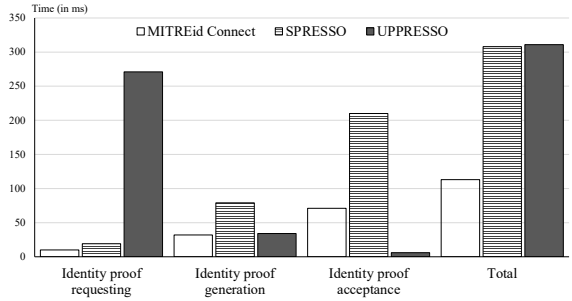

Figure 7: The Evaluation.

Setting. We compare UPPRESSO with MITREid Connect [**?**] and SPRESSO [**?**], where MITREid Connect provides open-source Java implementations [**?**] of IdP and RP's SDK and SPRESSO provides the JavaScript implementations based on node.js for all entities [**?**]. 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 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 $1,000$ 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 **??**), the RP (and user) constructing and transmitting the request to IdP; **Identity proof generation** (Steps 4.4-4.6 in Figure **??**), the IdP generating identity proof (no user authentication); and **Identity proof acceptance** (Steps 4.5-5.2 in Figure **??**), the RP server receives, verifies and parse the identity proof relayed from the IdP;

Results. The evaluation results are provided in Figure **??**. 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 in total. The significant overhead in UPPRESSO occurs when 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.

In the generation, UPPRESSO needs in total 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 to 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 $\mathcal{F}_{ID_U \mapsto PID_U}$, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ and $\mathcal{F}_{PID_U \mapsto Account}$ 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 [**?**] and user's identifier check [**?**]. With certificate transparency [**?**], 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 [**?**].

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 the 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 the 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) [**?**] and Kerberos [**?**]. These protocols are widely adopted in Google, Facebook, Shibboleth project [**?**], Java applications, 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 [**?**], SSO systems should prevent both RP-based identity linkage and IdP-based login tracing. The pairwise user identifier is adopted in SAML [**?**] and OIDC [**?**], and only prevents RP–based identity linkage; while SPRESSO [**?**] and BrowserID [**?**] only prevent IdP-based login tracing. BrowserID is adopted in Persona [**?**] and Firefox Accounts [**?**], however an analysis on Persona found IdP-based login tracing could still succeed [**?**, **?**]. UPPRESSO prevents both the RP-based identity linkage and IdP-based login tracing and could be integrated into OIDC which has been formally analyzed [**?**].

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 [**?**]. The notion of anonymous SSO was formalized [**?**] in 2013. And, various cryptographic primitives, such as group signature, zero-knowledge proof, etc., were adopted to design anonymous SSO schemes [**?**, **?**]. Anonymous SSO schemes are designed for the anonymous services, and not applicable to common services that need user identification.

Formal analysis on SSO standards. The SSO standards (e.g., SAML, OAuth, and OIDC) have been formally analyzed. Fett et al. [**?**, **?**] have conducted a formal analysis on OAuth 2.0 and OIDC standards based on an expressive Dolev-Yao style model [**?**], 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 the 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. [**?**, **?**] have proved that OAuth 2.0 and OIDC are secure once these two attacks are prevented. UPPRESSO could be integrated into OIDC, which simplifies its security analysis. [**?**] 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 of SSO implementations. Various vulnerabilities were found in SSO implementations, and then exploited for impersonation and identity injection attacks by breaking the confidentiality [**?**, **?**, **?**, **?**, **?**], integrity [**?**, **?**, **?**, **?**, **?**, **?**] or RP designation [**?**, **?**, **?**, **?**, **?**] of identity proof. Wang et al. [**?**] 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 [**?**], 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 [**?**, **?**, **?**, **?**, **?**], due to various vulnerabilities, such as XML Signature wrapping (XSW) [**?**], RP's incomplete verification [**?**, **?**, **?**], IdP spoofing [**?**, **?**] and etc. And, a dedicated, the bidirectional authenticated secure channel was proposed to improve the confidentiality and integrity of identity proof [**?**]. Vulnerabilities were also found to break the RP designation, such as the incorrect binding at IdPs [**?**, **?**], insufficient verification at RPs [**?**, **?**, **?**]. Automatical tools, such as SSOScan [**?**], OAuthTester [**?**] and S3KVetter [**?**], have been designed to detect vulnerabilities for breaking the confidentiality, integrity, or the 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 defends 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 of the prototype of UPPRESSO demonstrates that it supports an efficient SSO service, where a single login takes only 310 ms on average.

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

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

An HTTP response message is the term of the form

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

The details are defined as follows:

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

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

Origin. An Origin is a term $\langle host, protocol\rangle$ that 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 $\langle \texttt{POSTMESSAGE}, target, Content, Origin\rangle$, where `POSTMESSAGE` is the type, *target* is the constant nonce which stands for the receiver, *Content* is the message transmitted and *Origin* restricts the receiver'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 $\langle \texttt{XMLHTTPREQUEST}, URL, methods, Body, nonce\rangle$ can be converted into HTTP request message by the browser, and $\langle \texttt{XMLHTTPREQUEST}, Body, nonce\rangle$ is converted from HTTP response message.

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 $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, for instance the *body* in HTTP request is dictionary data. We assume there is a $body := \langle\langle username, alice\rangle, \langle password, 123\rangle\rangle$, here we define the operation on the *body*. That is, we can access the elements in *body* in the form $body[name]$, such that $body[username] \equiv alice$ and $body[password] \equiv 123$. We can also add the new attributes to the dictionary, for example after we set $body[age] := 18$, the *body* are changed into$\langle\langle username, alice\rangle, \langle password, 123\rangle, \langle age, 18\rangle\rangle$.

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 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 $\langle$`IFRAME`, *URL*, *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 $\langle ID, SignKey, sessions, users, RPs, Validity, Tokens\rangle$. 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 $\langle\langle Cookie, session\rangle\rangle$, the Cookie uniquely identifies the session and sessions store the browser uploaded messages.
- *users* is the set of user's information, including $username, password, ID_U$ and other attributes.
- *RPs* is the set of RP information which consists of ID of RP (PID_{RP}), $Endpoints$ (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_U.
- `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 **??** in Appendix **??**.

A.3.2 RP process

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

- ID_{RP} and $Endpoints$ are RP's registered information at IdP.
- *Cert* is the IdP signed RP information containing $ID_{RP}, Endpoints$ and other attributes.
- *IdP* is the term of the form $\langle ScriptUrl,\ q,\ PubKey\rangle$, 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 $a^{-1} \mod q$.
- `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 **??** in Appendix **??**.

A.3.3 IdP scripting process

The state of IdP scripting process *scriptstate* is a term in the form $\langle IdPDomain, Parameters, q, refXHR\rangle$, 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 **??** Algorithm **??**.

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 **??** Algorithm **??**.

A.4 Proof of Theorem ??

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 **??** 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 **??** is converted into whether the UPPRESSO system meets the requirement in Definition **??**. However, as we consider the attacker initially does not know any honest user's cookie, the requirement of Definition **??** 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 **??** proves that the UPPRESSO system meets Requirement **??**.

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. 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 **??**, the *script_rp* does not send any cookie.
- P^r. According to Algorithm **??**, the P^r does not send any cookie.

Therefore, Lemma **??** is proved. □

To prove UPPRESSO system meets Requirement **??**, we need to know how the cookie can be set as $c(u,r)$. Based on Algorithm **??**, 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 **??**, 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 **??**, we can prove that RP script does not send any stored passwords.
- P^r. According to Algorithm **??**, it is easy to find out that RP server does not receive or send any stored passwords.
- *script_idp*. Based on Algorithm **??**, we can find that IdP script sends the user's password at Line 68. The target of this message is Url whose host is *IdPDoaim* set at Line 67. The *IdPDomian* 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 **??** and does not send this parameter to any other processes.
- P^i. Based on Algorithm **??**, 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. □

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 **??**. 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 **??**. Therefore, *Cert* cannot be forged or modified.

- *RegistrationResult* is used in Algorithm **??** 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 **??** 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. □

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

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 **??**. 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 **??** 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 **??**, 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_U at Line 61, the ID_U 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_U . The ID_U 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 **??**, the IdP script can only receive the *Token* at Line 99 in Algorithm **??**. 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_{RP} in $t(u,r)$ as p. That is the PID_{RP} at Line 69 in Algorithm **??** must equal to p and the PID_{RP} is verified at Line 44 with the *RegistrationToken*. This verification cannot be bypassed due to the state check at Line 60. At the same validity period, the IdP script needs to send the registration request with same p and receive the successful registration result. As the IdP checks the uniqueness of PID_{RP} at Line 32 in Algorithm **??**. The r and IdP script must share the same *RegistrationToken*. As the *RegistrationToken* contains the $\texttt{Hash}(N_U)$, 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. □

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

Then, we prove that UPPRESSO meets Requirement **??**. 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 **??**, 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 **??**, 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 **??**, 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 **??**, 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 **??**. 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 **??**, PID_{RP} is valid at RP server only when RP server receives the registration result. The $\texttt{Hash}(N_U)$ 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 **??**, the registration result received by IdP script at Line 35 in Algorithm **??** 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 $\texttt{Hash}(N_U)$ 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.

□

Therefore, Requirement **??** is satisfied and Theorem **??** is proved.

B Algorithm

B.1 IdP process

Algorithm 1 R^i

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

let $s := s'$
let $n, method, path, parameters, headers, body$ **such that**
$\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$
if *possible*; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{IdPScript} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /login$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := body[username]$
  let $password := body[password]$
  if $password \not\equiv \texttt{SecretOfID}(username)$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure} \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $session[uid] := \texttt{UIDOfUser}(username)$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /loginInfo$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := session[username]$
  if $username \not\equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Logged} \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unlogged} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /dynamicRegistration$ **then**
  let $PID_{RP} := body[PID_{RP}]$
  let $Endpoint := body[Endpoint]$
  let $Nonce := body[Nonce]$
  if $PID_{RP} \in \texttt{ListOfPID}()$ **then**
    let $Content := \langle Fail, PID_{RP}, Nonce \rangle$
    let $Sig := \texttt{Sig}(Content, s'.SignKey)$
    let $RegistrationResult := \langle Content, Sig \rangle$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $Validity := \texttt{CurrentTime}() + s'.Validity$
  let $s'.RPs := s'.RPs +^{\langle\rangle} \langle PID_{RP}, Endpoint, Validity \rangle$
  let $Content := \langle OK, PID_{RP}, Nonce, Validity \rangle$
  let $Sig := \texttt{Sig}(Content, s'.SignKey)$
  let $RegistrationResult := \langle Content, Sig \rangle$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, RegistrationResult \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /authorize$ **then**

47: **let** $cookie := headers[Cookie]$
48: **let** $session := s'.sessions[cookie]$
49: **let** $username := session[username]$
50: **if** $username \equiv \texttt{null}$ **then**
51: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
52: **stop** $\langle f, a, m' \rangle, s'$
53: **end if**
54: **let** $PID_{RP} := parameters[PID_{RP}]$
55: **let** $Endpoint := parameters[Endpoint]$
56: **if** $PID_{RP} \notin \texttt{ListOfPID}() \vee Endpoint \notin \texttt{EndpointsOfRP}(PID_{RP})$ **then**
57: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
58: **stop** $\langle f, a, m' \rangle, s'$
59: **end if**
60: **let** $ID_U := session[uid]$
61: **let** $PID_U := \texttt{Multiply}(PID_{RP}, ID_U)$
62: **let** $Validity := \texttt{CurrentTime}() + s'.Validity$
63: **let** $Content := \langle PID_{RP}, PID_U, s'.ID, Validity \rangle$
64: **let** $Sig := \texttt{Sig}(Content, s'.SignKey)$
65: **let** $Token := \langle Content, Sig \rangle$
66: **let** $s'.Tokens := s'.Tokens +^{\langle\rangle} Token$
67: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Token, Token \rangle \rangle$
68: **stop** $\langle f, a, m' \rangle, s'$
69: **end if**
70: **stop** $\langle\rangle, s'$

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**
$\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$
if *possible*; **otherwise** stop $\langle\rangle, s'$
3: **if** $path \equiv /script$ **then**
4: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript} \rangle$
5: **stop** $\langle f, a, m' \rangle, s'$
6: **else if** $path \equiv /login$ **then**
7: **let** $m' := \langle \texttt{HTTPResp}, n, 302, \langle \langle Location, s'.IdP.ScriptUrl \rangle \rangle, \langle\rangle \rangle$
8: **stop** $\langle f, a, m' \rangle, s'$
9: **else if** $path \equiv /startNegotiation$ **then**
10: **let** $cookie := headers[Cookie]$
11: **let** $session := s'.sessions[cookie]$
12: **let** $N_U := parameters[N_U]$
13: **let** $PID_{RP} := \texttt{Multiply}(s'.ID_{RP}, N_U)$
14: **let** $T := \texttt{ExEU}(N_U, s'.IdP.q)$
15: **let** $session[N_U] := N_U$
16: **let** $session[PID_{RP}] := PID_{RP}$
17: **let** $session[t] := T$
18: **let** $session[state] := expectRegistration$
19: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Cert, s'.Cert \rangle \rangle$
20: **stop** $\langle f, a, m' \rangle, s'$
21: **else if** $path \equiv /registrationResult$ **then**
22: **let** $cookie := headers[Cookie]$
23: **let** $session := s'.sessions[cookie]$

```
    if session[state] ≢ expectRegistration then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨f, a, m′⟩, s′
    end if
    let RegistrationResult := body[RegistrationResult]
    let Content := RegistrationResult.Content
    if checksig(Content, RegistrationResult.Sig, s′.IdP.PubKey) ≡ FALSE then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        let session := null
        stop ⟨f, a, m′⟩, s′
    end if
    if Content.Result ≢ OK then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        let session := null
        stop ⟨f, a, m′⟩, s′
    end if
    let PID_RP := session[PID_RP]
    let N_U := session[N_U]
    let Nonce := Hash(N_U)
    let Time := CurrentTime()
    if PID_RP ≢ Content.PID_RP ∨ Nonce ≢ Content.Nonce ∨ Time > Content.Validity then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        let session := null
        stop ⟨f, a, m′⟩, s′
    end if
    let session[PIDValidity] := Content.Validity
    let Endpoint ∈ s′.Endpoints
    let session[state] := expectToken
    let Nonce′ := Random()
    let session[Nonce] := Nonce′
    let Body := ⟨PID_RP, Endpoint, Nonce′⟩
    let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Body⟩
    stop ⟨f, a, m′⟩, s′
else if path ≡ /uploadToken then
    let cookie := headers[Cookie]
    let session := s′.sessions[cookie]
    if session[state] ≢ expectToken then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨f, a, m′⟩, s′
    end if
    let Token := body[Token]
    if checksig(Token.Content, Token.Sig, s′.IdP.PubKey) ≡ FALSE then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨f, a, m′⟩, s′
    end if
    let PID_RP := session[PID_RP]
    let Time := CurrentTime()
    let PIDValidity := session[PIDValidity]
    let Content := Token.Content
    if PID_RP ≢ Content.PID_RP ∨ Time > Content.Validity ∨ Time > PIDValidity then
        let m′ := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨f, a, m′⟩, s′
    end if
    let PID_U := Content.PID_U
```

78: **let** $T := session[t]$
79: **let** $Account := \texttt{Multiply}(PID_U, T)$
80: **if** $Account \in \texttt{ListOfUser}()$ **then**
81: **let** $\texttt{RegisterUser}(Account)$
82: **end if**
83: **let** $session[user] := Account$
84: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSuccess} \rangle$
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** $target := \texttt{PARENTWINDOW}(tree, docnonce)$
4: **let** $IdPDomain := s'.IdPDomain$
5: **switch** $s'.q$ **do**
6: **case** *start*:
7: **let** $N_U := \texttt{Random}()$
8: **let** $command := \langle \texttt{POSTMESSAGE}, target, \langle \langle N_U, N_U \rangle \rangle, \texttt{null} \rangle$
9: **let** $s'.Parameters[N_U] := N_U$
10: **let** $s'.q := expectCert$
11: **case** *expectCert*:
12: **let** $pattern := \langle \texttt{POSTMESSAGE}, *, Content, * \rangle$
13: **let** $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
14: **if** $input \not\equiv \texttt{null}$ **then**
15: **let** $Cert := input.Content[Cert]$
16: **if** $\texttt{checksig}(Cert.Content, Cert.Sig, s'.PubKey) \equiv \texttt{null}$ **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** $PID_{RP} := \texttt{Multiply}(Cert.Content.ID_{RP}, N_U)$
22: **let** $s'.Parameters[PID_{RP}] := PID_{RP}$
23: **let** $Endpoint := \texttt{RandomUrl}()$
24: **let** $s'.Parameters[Endpoint] := Endpoint$
25: **let** $Nonce := \texttt{Hash}(N_U)$
26: **let** $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /dynamicRegistration, \langle\rangle \rangle$
27: **let** $s'.refXHR := \texttt{Random}()$
28: **let** $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{POST},$
$\langle \langle PID_{RP}, PID_{RP} \rangle, \langle Nonce, Nonce \rangle, \langle Endpoint, Endpoint \rangle \rangle, s'.refXHR \rangle$
29: **let** $s'.q := expectRegistrationResult$
30: **end if**
31: **case** expectRegistrationResult:
32: **let** $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$
33: **let** $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
34: **if** $input \not\equiv \texttt{null} \wedge input.Content[RegistrationResult].type \equiv OK$ **then**
35: **let** $RegistrationResult := input.Body[RegistrationResult]$
36: **if** $RegistrationResult.Content.Result \not\equiv OK$ **then**
37: **let** $s'.q := stop$
38: **let stop** $\langle\rangle$

```
39:         end if
40:         let command := ⟨POSTMESSAGE, target, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, null⟩
41:         let s'.q := expectProofRquest
42:       end if
43:     case expectProofRquest:
44:       let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
45:       let input := CHOOSEINPUT(scriptinputs, pattern)
46:       if input ≠ null then
47:         let PID_RP := input.Content[PID_RP]
48:         let Endpoint_RP := input.Content[Endpoint]
49:         let s'.Parameters[Nonce] := input.Content[Nonce]
50:         let Cert := s'.Parameters[Cert]
51:         if Endpoint_RP ∉ Cert.Content.Endpoints ∨ PID_RP ≠ s'.Parameters[PID_RP] then
52:           let s'.q := stop
53:           let stop ⟨⟩
54:         end if
55:         let s'.Parameters[Endpoint_RP] := Endpoint_RP
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 PID_RP := s'.Parameters[PID_RP]
72:           let Endpoint := s'.Parameters[Endpoint]
73:           let Nonce := s'.Parameters[Nonce]
74:           let Url := ⟨URL, S, IdPDomain, /authorize,
                ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩
75:           let s'.refXHR := Random()
76:           let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
77:           let s'.q := expectToken
78:         end if
79:       end if
80:     case expectLoginResult:
81:       let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
82:       let input := CHOOSEINPUT(scriptinputs, pattern)
83:       if input ≠ null then
84:         if input.Body ≠ LoginSuccess then
85:           let stop ⟨⟩
86:         end if
87:         let PID_RP := s'.Parameters[PID_RP]
88:         let Endpoint := s'.Parameters[Endpoint]
89:         let Nonce := s'.Parameters[Nonce]
90:         let Url := ⟨URL, S, IdPDomain, /authorize,
              ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩⟩
```

```
91:          let s'.refXHR := Random()
92:          let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s'.refXHR⟩
93:          let s'.q := expectToken
94:        end if
95:      case expectToken:
96:        let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
97:        let input := CHOOSEINPUT(scriptinputs, pattern)
98:        if input ≠ null then
99:          let Token := input.Body[Token]
100:         let RPOringin := ⟨s'.Parameters[Endpoint_RP], S⟩
101:         let command := ⟨POSTMESSAGE, target, ⟨Token, Token⟩, RPOrigin⟩
102:         let s.q := stop
103:       end if
104: end switch
105: let stop ⟨s', cookies, localStorage, sessionStorage, command⟩
```

B.4 RP scripting process

Algorithm 4 *script_rp*

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

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