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. Our analysis shows UPPRESSO provides a comprehensive privacy protection while achieving the same security guarantees of OIDC.

Keywords*—*Single sign-on, security, privacy.

I. Introduction

Single sign-on (SSO) systems, such as OpenID Connect [**?**], OAuth [**?**] and SAML [**?**], have been widely deployed as the identity management and authentication infrastructure in the Internet. SSO enables a website, called the *relying party* (RP), to delegate its user authentication to a trusted third party called the *identity provider* (IdP). Thus, a user visits multiple RPs with only a single explicit authentication attempt at the IdP. With the help of SSO, a user no longer needs to remember multiple credentials for different RPs; instead, she maintains only the user credential for the IdP, which generates *identity proofs* for her visits to these RPs. SSO has been widely integrated with many application services. For example, we find that 80% of the Alexa Top-100 websites [**?**] support SSO, and the analysis on the Alexa Top-1M websites [**?**] identifies 6.30% with the SSO support. Meanwhile, many email and social network providers (such as Google, Facebook, Twitter, etc.) are serving the IdP roles in the Internet.

The adoption of SSO also raises privacy concerns regarding online user tracking and profiling [**?**], [**?**]. User privacy leaks in all existing SSO protocols and implementations. Taking a widely used SSO protocol, OpenID Connect (OIDC), as an example, we explain its login process and the risk of user privacy leakage. On receiving a user's login request, the RP constructs a request of identity proof with its identity and redirects it to the IdP. After authenticating the user, the IdP generates an identify proof binding the identities of the user and the RP, which is forwarded to the RP by the user. Finally, the RP verifies the identity proof and allows the user to log in. From such login instances, any curious IdP or collusive RPs could break the users' privacy as follows.

- *IdP-based login tracing*. The IdP knows the identities of the RP and the user in each single login instance, to generate the identity proof. As a result, a curious IdP could discover all the RPs that the victim user attempts to visit and profile her online activities.
- *RP-based identity linkage*. The RP learns a user's identity from the identify proof. When the IdP generates identity proofs for a user, if the same user identifier is bound in identity proofs generated for different RPs, which is the case of several widely deployed SSO systems [**?**], [**?**], malicious RPs could collude to not only link the user's login activities at different RPs for online tracking but also associate her attributes across multiple RPs [**?**].

Large IdPs, especially the social IdPs such as Google and Facebook, are interested in collecting users' online behavioral information for various purposes (e.g., Screenwise Meter [**?**] and Onavo [**?**]). By simply serving the IdP role, these companies will easily collect a large amount of data to reconstruct users' online traces. On the other hand, in the Internet, many service providers host a variety of web services and therefore take an advantaged position to link a user's multiple logins at different RPs. Through the internal information integration, rich information will be obtained from the SSO data for user profiling. Meanwhile, the technologies of privacy-preserving record linkage [**?**] and private set intersection [**?**] allow multiple organizations (e.g., RPs) to share and link the data without directly sharing their clients' data, which pave the path for cross-organizational RP-based identity linkage.

While the privacy problems in SSO have been widely recognized [**?**], [**?**], only a few solutions were proposed to protect user privacy [**?**], [**?**]. Among them, Pairwise Pseudonymous Identifier (PPID) [**?**], [**?**] is a straightforward and commonly adopted solution to defend against RP-based identity linkage. It requires the IdP to create different identifiers for the user when she logs into different RPs, so that the pseudo-identifiers of the same user cannot be used to link the user's logins at different RPs even if they collude. As a recommended practice

by NIST [**?**], PPID has been specified in many widely adopted SSO standards including OIDC [**?**] and SAML [**?**]. However, PPID-based approaches cannot prevent the IdP-based login tracing attacks, as the IdP still knows which RP the user visits.

To the best of our knowledge, only two schemes (i.e., BrowserID [**?**] and SPRESSO [**?**]) have been proposed so far to defend against IdP-based login tracing. In BrowserID (and its prototypes known as Mozilla Persona [**?**] and Firefox Accounts [**?**]), the IdP generates a special "identity proof" to bind the user's unique identifier (i.e., email address) to a public key. With the corresponding private key, the user signs an extra subsidiary identity proof to bind the visited RP's identity and its identity, and sends this pair of identity proofs to that RP. In this way, the IdP does not know the RP's identity when generating identify proofs. SPRESSO requires the RP to create a one-time pseudo-identifier at each login for the IdP to generate an identity proof, and then hides the RP's real identity from the IdP. The RP employs a third-party entity called forwarder, which works as a proxy to relay the identity proof from the IdP to the corresponding RP. In both schemes, the RPs' identities are protected from the IdP; however, the IdP has to know the user's unique identifier (e.g., email address) and includes it in identity proofs so that the visited RP can recognize the user in her multiple logins. As a result, both schemes are still vulnerable to RP-based identity linkage.

As discussed above, none of the existing SSO systems defend against both IdP-based login tracing and RP-based identity linkage at the same time. Before presenting our solution, we first formally analyze the privacy problems and solutions. Let us denote the user's and the visited RP's identities as ID_U and ID_{RP}, respectively. To protect user privacy against RP-based identity linkage, ID_U should not be explicitly included in the identity proof which will be received by the RP. Instead, a privacy-preserving pseudo-identifier PID_U should be used (as in the PPID-based approaches [**?**], [**?**]), which can be viewed as the output of a one-way identifier-transformation function $\mathcal{F}_{ID_U \mapsto PID_U}$ at the IdP, which authenticates the user and then knows ID_U. Similarly, to prevent IdP-based login tracing, ID_{RP} should not be explicitly included in the identity proof but be replaced by a pseudo-identifier PID_{RP} (as in SPRESSO [**?**] and BrowserID [**?**]), which is obtained by another one-way function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ at the RP. However, if both PID_U and PID_{RP} are used in identity proofs to replace ID_U and ID_{RP} at the same time, assuming they can be securely exchanged between the IdP and the RP in an SSO login process, the RP will allow the user to log in as PID_U, which will be different in the user's multiple logins at a same RP; otherwise, the IdP might be able to associate these RPs visited by the user. As a result, the RP has no clues to derive the user account at the RP but has to treat her as a one-time user every time when she logs in. This violates the basic requirements of SSO services.

In this paper, we propose an Unlinkable Privacy-PREserving Single Sign-On (UPPRESSO) system to provide comprehensive protections against both IdP-based login tracing and RP-based identity linkage. The key idea of UPPRESSO is to design a special identifier-transformation function $\mathcal{F}_{PID_U \mapsto Account}$, which maps *all* PID_Us of a user to *one* unique $Account$ in all logins to an RP, where $Account$ is created when the user logs into the RP for the first time. Since in every login instance PID_U and PID_{RP} are separately generated by the IdP and the RP, respectively, we have to design associative one-way identifier-transformation functions $\mathcal{F}_{ID_U \mapsto PID_U}$ and $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$, so that three identifier-transformation functions work cooperatively to ensure: (*a*) when a user logs into an RP for multiple times, the RP always maps PID_Us to an identical $Account$ without knowing the user's identity; moreover, when a user logs into multiple RPs, (*b*) a curious IdP cannot link multiple PID_{RP}s to a particular RP or associate them together, and collusive RPs (*c*) cannot link PID_Us to a particular user or associate them together, (*d*) nor link $Account$s of a same user at different RPs.

To completely achieve these goals, we design three one-way identifier-transformation functions based on the discrete logarithm problem. We design a one-way trapdoor function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}(ID_{RP}, T)$ for an RP to generate a random PID_{RP} based on a randomly generated trapdoor T, and a one-way function $\mathcal{F}_{ID_U \mapsto PID_U}(ID_U, PID_{RP})$ for the IdP to generate PID_U based on PID_{RP}. With the trapdoor T, the RP applies $\mathcal{F}_{PID_U \mapsto Account}(PID_U, PID_{RP}, T)$ to identity proofs binding PID_{RP} and PID_U, to derive the unique $Account$. We summarize our contributions as follows.

- We formally analyze the privacy problems in SSO as an identifier-transformation problem, and propose the first comprehensive solution to hide the users' login traces from both the curious IdP and malicious collusive RPs. To the best of our knowledge, UPPRESSO is the first SSO system that provides secure SSO services against IdP-based login tracing and RP-based identity linkage.
- We systematically analyze the security of UPPRESSO and show that it achieves the same security level as existing SSO systems, while the users' login traces are well protected.
- We have implemented a prototype of UPPRESSO based on an open-source implementation of OIDC, which requires only small modifications to support three identifier-transformation functions for privacy protections. Unlike BrowserID and SPRESSO, UPPRESSO does not require non-trivial re-designs of SSO services, which makes it more compatible with existing SSO systems.
- We compare the performance of the 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 first introduce the background and preliminaries in Section II. Then, we describe the identifier-transformation based approach, the threat model, and our UPPRESSO design in Sections III, IV and V, followed by a systematical analysis of security and privacy in Section **??**. We provide the implementation specifics and experiment evaluation in Section VII, discuss the extensions and related works in Section VIII and IX, and conclude our work in Section X.

II. Background and Preliminaries

UPPRESSO is designed to be compatible with OpenID Connect (OIDC) and provide privacy protections based on

the discrete logarithm problem. Here, we provide a brief introduction about OIDC and the discrete logarithm problem.

A. OpenID Connect (OIDC)

As an extension of OAuth 2.0 for user authentication, OIDC is one of the most popular SSO protocols [**?**]. As other SSO protocols [**?**], OIDC involves three entities, i.e., *users*, the *identity provider (IdP)*, and *relying parties (RPs)*. Both users and RPs register at the IdP with identifiers (ID_U, ID_{RP} and PID_U in some schemes), and the necessary information such as credentials, RP endpoints (e.g., the URLs to receive identity proofs), etc. The IdP is assumed to maintain these attributes securely. In an OIDC login process, a user is responsible for initiating a login request at an RP, redirecting the SSO messages between the RP and the IdP, and checking the scope of user attributes in the identify proof generated by the IdP for the visited RP. Usually, the redirection and checking actions are handled by a user-controlled software, known as *user agent* (e.g., browser). Once receiving a user login request, the RP constructs an identity proof request with its identifier and the scope of requested user attributes, sends the identity proof request to the IdP through the user, and parses the received identity proof to authenticate the user. The IdP authenticates the user based on her ID_U and credential, maps ID_U to PPID (i.e., privacy-preserving pseudo-identifier) based on the RP identity (i.e., ID_{RP}), generates an identity proof containing PPID, ID_{RP} and requested user attributes, and returns the identity proof to the endpoint registered by the RP.

OIDC Implicit Flow. OIDC supports three user login flows, which are the *implicit flow*, *authorization code flow* and *hybrid flow* (i.e., a mix-up of the previous two). In the implicit flow, an *id token* is generated as the identity proof, which contains a user identifier, an RP identifier, the issuer (i.e., IdP), the validity period, and other requested attributes. The IdP signs the id token using its private key to ensure integrity, and sends it to the RP through the user. In the authorization code flow, the IdP binds an authorization code with the RP, and sends it to the RP through the user; then, the RP establishes an HTTPS connection to the IdP and uses the authorization code with the RP's credential to obtain the user's identifier and other attributes. UPPRESSO is compatible with all three flows. For brevity, we will present our design and implementation of UPPRESSO on top of the OIDC implicit flow in details, and discuss the extension to support the authorization code flow in Section VIII.

The original OIDC implicit flow is shown in Figure 1. When a user attempts to log into an RP, the RP constructs an identity proof request and returns it to the user, which gets redirected to the IdP. The request contains ID_{RP}, the RP endpoint to receive the identity proof, and the scope of requested user attributes. If the user has not yet been authenticated, the IdP initiates an authentication process to authenticate her. For a successfully authenticated user, the IdP generates an id token and returns it to the RP endpoint. If the endpoint is not registered for that RP, the IdP will return a warning to notify the user about potential identity proof leakage. Once the RP receives the identity proof, it makes the authentication decision after verifying the identity proof.

RP Dynamic Registration. The RP dynamic registration [**?**] of OIDC allows an RP to update its information at the IdP. When an RP first registers at the IdP, it obtains a registration token, with which the RP can initiate a dynamic registration process to update its information (e.g., the endpoint). After a successful dynamic registration, the RP obtains a new unique ID_{RP} from the IdP. UPPRESSO leverages this function and slightly modify the dynamic registration process to enable PID_{RP} *registration*, which allows an RP to generate different privacy-preserving identifiers (PID_{RP}s) and register them at the IdP.

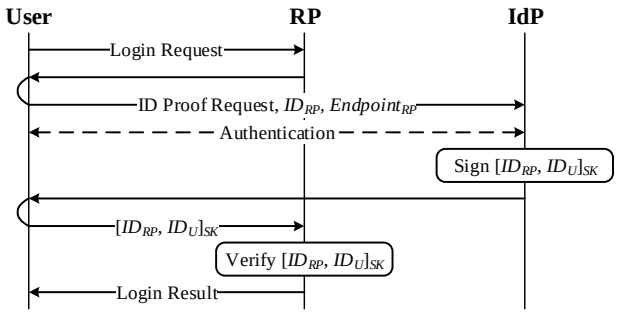

Fig. 1: The implicit flow of OIDC.

B. Discrete Logarithm Problem

Based on the discrete logarithm problem, UPPRESSO designs the identifier-transformation functions. Here, we briefly review the discrete logarithm problem. For the finite field $GF(p)$ where p is a large prime, a number g is called a generator of order q, if it constructs a cyclic group of q elements by calculating $y = g^x \ mod \ p$. And x is called the discrete logarithm of y modulo p. Given a large prime p, a generator g and a number y, it is computationally infeasible to solve the discrete logarithm (i.e., x) of y [**?**], which is called the discrete logarithm problem. The hardness of solving discrete logarithms is utilized to design several secure cryptographic primitives, including Diffie-Hellman key exchange and the digital signature algorithm (DSA).

III. The Privacy Dilemma and UPPRESSO Overview

In this section, we first review the security and privacy requirements of SSO services and discuss the privacy dilemma in designing secure privacy-preserving SSO systems. Then, we provide an overview of the identifier-transformation approach of UPPRESSO.

A. Security Requirements of SSO

The primary goal of SSO services is to implement secure user authentication [**?**], i.e., to ensure that a legitimate user can always log into an honest RP under her unique account. To achieve this, every identity proof generated by the IdP should explicitly specify the user who is authenticated by the IdP (i.e., ***user identification***) and the RP which the user attempts to log into (i.e., ***RP designation***). User identification also requires an RP to be able to recognize every authenticated user under a unique account registered at that RP in multiple logins of the user. Moreover, the identify proof should be generated by

the trusted IdP and correctly transmitted to the dedicated RP and the user in some schemes (i.e., ***confidentiality***). In this process, the identify proof should not be modified or forged (i.e., ***integrity***). These are the basic security requirements of SSO systems summarized based on theoretical analysis of SSO designs [**?**], [**?**], [**?**] and practical attacks [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].

Many attacks exploit vulnerabilities in SSO design or implementation to break at least one of these security requirements, so that the adversaries can log into an honest RP as a victim user (called *impersonation attacks*), or allure a victim user to log into 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 could replay the identity proof received from a user for itself and log into Friendcaster as the victim user. Or, the adversaries could impersonate the victim user with a leaked identity proof (i.e., a violation of confidentiality) [**?**], [**?**], [**?**], [**?**], [**?**]. It was also reported that some RPs of Google ID SSO accepted user attributes that were not tied to the identity proof (i.e., a potential violation of integrity) [**?**]. As a result, an adversary could insert arbitrary attributes (e.g., an email address) into the identity proof to impersonate another user at the RP.

B. The Privacy Dilemma in SSO Identity Proofs

As discussed in Section I, existing SSO systems are vulnerable to IdP-based login tracing or RP-based identity linkage privacy leakage. We argue that a secure and privacy-preserving SSO system should prevent *both* types of privacy leakage while satisfying *all* four basic security requirements. However, meeting the security and privacy requirements at the same time incurs a dilemma in the identity proof generation.

In an SSO login instance, the identity proof is generated by the IdP about the authenticated user and the requesting RP. First, to prevent IdP-based login tracing, the identity proof request should not disclose ID_{RP} to the IdP, since the IdP already knows ID_U after authenticating the user. However, to ensure RP designation, the IdP should bind each identify proof with a certain transitional pseudo-identifier of the RP (denoted as PID_{RP}). PID_{RP} can be generated by the user or the RP, or together, but it should be computationally infeasible for the IdP to derive ID_{RP} from PID_{RP}.

Meanwhile, to prevent RP-based identity linkage, the identity proof should not directly contain ID_U or disclose it in any means. This requires the IdP to generate a certain transitional pseudo-identifier of the user (denoted as PID_U) and bind it to the identity proof. However, to ensure user identification, the requesting RP should be able to recognize the user and associate PID_U to her unique account at the RP (denoted as $Account$). Similarly, it should be computationally infeasible for the RP to derive ID_U from PID_U or $Account$.

We illustrate the relationships between the identifiers, pseudo-identifiers, and the identity proof in Figure 2, where red and green blocks respectively represent long-term identities and one-time pseudo-identifiers known to each entity, and the arrows denote how the pseudo-identifiers are obtained. A *dilemma* exists in the SSO login process, that is, the IdP is expected to generate a PID_U for the authenticated user ID_U that can be linked to her $Account$ at the requesting RP, *without knowing any information about the requesting RP.*

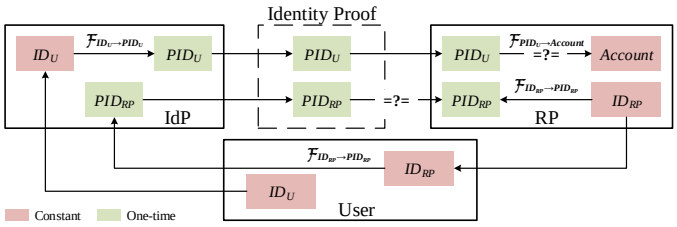

Fig. 2: Identifier transformations in privacy-preserving SSO.

C. The Identifier-transformation Framework and UPPRESSO

As shown in Figure 2, we convert the privacy-preserving SSO login problem into an identifier-transformation problem, which aims to design three identifier-transformation functions $\mathcal{F}_{ID_U \mapsto PID_U}$, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$, and $\mathcal{F}_{PID_U \mapsto Account}$ to compute PID_U, PID_{RP} and $Account$ that satisfy the desired security and privacy requirements. To solve the dilemma, the key is to pass some information related to the user's $Account$ at the RP to the IdP to assist its generation of PID_U, so that all PID_Us of the same user during her multiple logins at the same RP can be correctly linked to her $Account$. Meanwhile, such information should not provide any additional knowledge for the IdP to infer the RP's identity (i.e., ID_{RP}).

To achieve this goal, UPPRESSO constructs three transformation functions in an integrated way to support *transformed RP designation* and *trapdoor user identification*, where (i) PID_Us and PID_{RP}s are generated dynamically for different logins to protect user privacy; (ii) in each login, PID_{RP} helps to link PID_U to $Account$; and (iii) in a user's multiple logins to the same RP, an invariant $Account$ can always be derived from different pairs of PID_Us and PID_{RP}s.

Transformed RP designation. To prevent IdP-based login tracing, the identity proof request should not include ID_{RP} but a PID_{RP} dynamically transformed from ID_{RP}. 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 for each login. Then, the user assists the RP to register this PID_{RP} at the IdP (through OIDC dynamic registration). When an RP receives an identity proof, it can verify if the enclosed PID_{RP} is transformed from its own ID_{RP} using the trapdoor it holds.

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 and include it in the identity proof. 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.

TABLE I: 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.

D. Existing Privacy-Preserving SSO Solutions

Next, we map three existing privacy-preserving SSO approaches (PPID [**?**], [**?**], BrowserID [**?**] and SPRESSO [**?**]) to the identifier-transformation framework in Figure 2 and explain their designs and potential privacy issues as shown in Table I. 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, the IdP generates different PID_Us for a user to log into different RPs and maintains deterministic one-to-many mappings from ID_U to PID_Us. Therefore, RP-based identity linkage is prevented. At each RP, $Account = PID_U$ ensures user identification. However, since ID_{RP} is directly used in identity proofs ($PID_{RP} = ID_{RP}$), they cannot defend against IdP-based login tracing.

In SPRESSO, the RP generates PID_{RP} by encrypting ID_{RP} padded with a nonce for each login session and forwards PID_{RP} to the IdP through the user. With the corresponding nonce, the RP can verify the PID_{RP} in the identity proof to ensure RP designation, while hiding ID_{RP} from the IdP to defend against IdP-based login tracing. However, the IdP uses a constant ID_U to generate identity proofs for each user regardless of which RP she requests to login to. As a result, SPRESSO is vulnerable to RP-based identity linkage, since $Account$ always equals to ID_U.

The identity proofs in the BrowserID system include ID_U but no RP information (i.e., $PID_{RP} = \perp$), therefore, it prevents IdP-based login tracing. To ensure RP designation, BrowserID requires the user to append a *subsidiary* identity proof and sign it, where the identity proof signed by the IdP authorizes the user to sign the subsidiary identity proof. Obviously, ID_U is tied to a pair of identity proof and subsidiary identity proof and passed to the RP. Therefore, $Account = ID_U$, which makes BrowserID vulnerable to RP-based identity linkage.

As we can see, none of these approaches prevents 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 arbitrarily designed and function separately, which causes either $PID_{RP} = ID_{RP}$ or $Account = ID_U$.

IV. Threat Model and Assumptions

Similar as other SSO systems (e.g., SAML and OIDC), UPPRESSO consists of an IdP and multiple RPs and users. Next, we describe the threat model and some assumptions.

A. 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 specifications, but may be interested in users' login activities at multiple RPs. It is assumed to be well-protected and never leaks sensitive information. For example, the IdP is trusted to maintain the private key for signing identity proofs and RP certificates (see Section V-B for details), so that the adversaries can never forge an identity proof or an RP certificate. An honest IdP should never collude with other entities (e.g., malicious RPs or users), while a curious IdP tries to break user privacy without violating the protocol. For example, it may store all the received messages to infer the relationship among ID_U, ID_{RP}, PID_U and PID_{RP} and trace a user's login activities.

Malicious Users. We assume the adversary can control a set of users, by stealing users' credentials [**?**], [**?**] or directly registering sybil accounts at the IdP and RPs, to break the security of UPPRESSO. They may impersonate a benign user at honest RPs, or trick a victim user to log into an honest RP under the account of a malicious user. For example, the malicious users 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 could also control a set of RPs 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. For example, a malicious RP may manipulate its PID_{RP} and trick the users to submit identity proofs for other honest RPs to itself and reply them, 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 together, attempting to break the security and privacy guarantees. For example, malicious users and RPs may manipulate PID_U and PID_{RP} in an identity proof collusively to perform the impersonation or identity injection attacks.

B. Assumptions

We assume the user agent deployed at honest users is correctly implemented and transmits messages to the dedicated receivers as expected. We also assume the IdP and benign RPs and users adopt TLS to ensure confidentiality and integrity in the communications. Meanwhile, the cryptographic algorithms (such as RSA and SHA-256) and building blocks (such as random number generators and the discrete logarithm problem) used in UPPRESSO are assumed to be correctly implemented.

In UPPRESSO, we study the RP-based identity linkage caused by a same user identifier used across different RPs. While the RPs may be able to re-identify a user from some distinctive user attributes, such as telephone numbers, addresses, or driver licenses, we consider it out of the scope of this paper.

Also, we focus on IdP-based login tracing attacks that are enabled by SSO protocols, but do not consider other network attacks such as traffic analysis that trace a user's logins at different RPs.

V. The Design of UPPRESSO

Next, we present the main design of UPPRESSO. First, we describe our design for the three identifier-transformation functions and the details of the transformed RP designation and trapdoor user identification schemes. Then, we provide an overview of the UPPRESSO system and its login flow. Finally, we discuss the compatibility of UPPRESSO with OIDC. The notations used in this paper are listed in Table II.

A. Identifier-transformation Functions in UPPRESSO

As discussed in Section III, the identifier-transformation functions are essential for privacy-preserving SSO systems. In UPPRESSO, the three functions, $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$, $\mathcal{F}_{ID_U\mapsto PID_U}$ and $\mathcal{F}_{PID_U\mapsto Account}$, are all constructed based on the discrete logarithm problem with public parameters p, q, and g, where p is a large prime defining the finite field $GF(p)$, q is a prime factor of $(p-1)$, and g is a generator of order q in $GF(p)$.

Without loss of generality, we assume the IdP assigns long-term identifiers ID_U to a user and ID_{RP} to an RP when they first register at the IdP. In particular, the IdP assigns a unique random number to each user as ID_U, where $1 < ID_U < q$. For each RP, the IdP selects a random number r, where $1 < r < q$, and computes a unique ID_{RP} as:

$$ID_{RP} = g^r \bmod p \tag{1}$$

where r is kept secret from the RP.

RP Identifier Transformation. In each login session, the user negotiates with the RP she tries to log in, and computes a pseudo-identifier PID_{RP} for the RP cooperatively. First, the RP chooses a random number N_{RP} $(1 < N_{RP} < q)$, and the user chooses another random number N_U $(1 < N_U < q)$. Then, they exchange N_{RP} and N_U to calculate PID_{RP} following Equation 2.

$$\mathcal{F}_{ID_{RP}\mapsto PID_{RP}} : PID_{RP} = {ID_{RP}}^{N_U N_{RP}} \bmod p \tag{2}$$

The transformation function $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$ satisfies the following requirements. First, it is computationally infeasible for the IdP to derive ID_{RP} from PID_{RP} due to the discrete logarithm problem. Moreover, two nonces N_U and N_{RP} ensure that: (*a*) PID_{RP} is valid only for this login and for the identity proof generated in this login, and (*b*) PID_{RP} is dynamically generated for this login and is different from other PID_{RP}s generated in other login session between the same user and RP. Therefore, the IdP cannot associate multiple PID_{RP}s of a same RP. Finally, the cooperative generation process between the user and the RP prevents a single malicious entity from manipulating the value of PID_{RP}.

User Identifier Transformation. When the IdP receives an identity proof request for ID_U and PID_{RP} from an authenticated user, it follows the transformation function in Equation 3 to calculate PID_U and includes it in the identity proof.

$$\mathcal{F}_{ID_U\mapsto PID_U} : PID_U = {PID_{RP}}^{ID_U} \bmod p \tag{3}$$

TABLE II: The notations used in UPPRESSO.

Notation	Definition	Attribute
p	A large prime.	Long-term constant
q	A large prime factor of $(p-1)$.	Long-term constant
g	A generator of order q in $GF(p)$.	Long-term constant
SK, PK	The private/public key of IdP.	Long-term constant
ID_{RP}	$ID_{RP} = g^r \bmod p$, an RP's unique identity.	Long-term constant
$Cert_{RP}$	An RP certificate, containing the RP's identity and endpoint.	Long-term constant
ID_U	A user's unique identity.	Long-term constant
$Account$	$Account = {ID_{RP}}^{ID_U} \bmod p$, a user's identifier at an RP.	Long-term constant
PID_{RP}	$PID_{RP} = {ID_{RP}}^{N_U N_{RP}} \bmod p$, an RP's pseudo-identifier.	One-time variable
PID_U	$PID_U = {PID_{RP}}^{ID_U} \bmod p$, a user's pseudo-identifier.	One-time variable
N_U	A user-generated nonce for PID_{RP}.	One-time variable
N_{RP}	An RP-generated nonce for PID_{RP}.	One-time variable
Y_{RP}	$Y_{RP} = {ID_{RP}}^{N_{RP}} \bmod p$, the public value for N_{RP}.	One-time variable
T	$T = (N_U N_{RP})^{-1} \bmod q$, the trapdoor to derive $Account$.	One-time variable

From Equations 1, 2 and 3, we see that $PID_U = {ID_{RP}}^{N_U N_{RP} ID_U} = g^{r N_U N_{RP} ID_U} \bmod p$. First, the discrete logarithm problem ensures that the RP cannot derive ID_U from PID_U.

Moreover, ID_{RP} is generated following Equation 1 to introduce a random r that is unknown to the RP, so that for a given ID_U, PID_U is determined by r, N_U and N_{RP} together. Otherwise, if two collusive RPs know r_1 and r_2 respectively, they can check if ${PID_{U_1}}^{r_2 N_{U_2} N_{RP_2}} = {PID_{U_2}}^{r_1 N_{U_1} N_{RP_1}} \bmod p$ holds, which means a same user logs into them in two SSO sessions (i.e., $ID_{U_1} == ID_{U_2}$).

User Account Transformation. The RP can derive a unique $Account$ for each user following Equation 4.

$$\mathcal{F}_{PID_U\mapsto Account} : Account = {PID_U}^T \bmod p \tag{4}$$

Here, we define $T = (N_U N_{RP})^{-1} \bmod q$ as the RP's trapdoor. As q is a prime number and $1 < N_U, N_{RP} < q$, q is coprime to $N_U N_{RP}$. Also, there always exists a T that satisfies $T(N_U N_{RP}) = 1 \bmod q$. Moreover, from Equations 2, 3 and 4, we have $Account = {ID_{RP}}^{ID_U} \bmod p$, derived as below.

$$\begin{aligned} Account &= {PID_U}^T = ({PID_{RP}}^{ID_U})^{(N_U N_{RP})^{-1} \bmod q} \\ &= {ID_{RP}}^{ID_U N_U N_{RP} T \bmod q} = {ID_{RP}}^{ID_U} \bmod p \end{aligned} \tag{5}$$

Therefore, when a user logins at an RP multiple times, the RP can always derives the same $Account$ to identity the user. Finally, the transformation function $\mathcal{F}_{PID_U\mapsto Account}$ satisfies the following requirements: (a) due to the discrete logarithm problem, the RP cannot derive ID_U from $Account$, and (b) collusive RPs cannot link a user's $Account$s at different RPs.

These three identifier-transformation functions enable *transformed RP designation* and *trapdoor user identification* to satisfy all the security and privacy requirements of an SSO.

Transformed RP Designation. $\mathcal{F}_{ID_{RP}\mapsto PID_{RP}}$ ensures that the user and RP cooperatively generate a fresh PID_{RP} in each login, while $\mathcal{F}_{ID_U\mapsto PID_U}$ ensures that the IdP generates the exact PID_U for the ID_U who logins at PID_{RP}. The

IdP will bind PID_U with PID_{RP} in the identity proof, which designates this identity proof to PID_{RP}. Finally, the transformed RP designation is provided through two phases. The function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ prevents the curious IdP from linking PID_{RP}s of different logins at an RP, and therefore avoids IdP-based login tracing.

Trapdoor User Identification. In a user's multiple logins, the RP expresses different PID_Us and has the corresponding Ts, so that derives the identical $Account$. The comprehensive design of identifier-transformation functions prevents collusive RPs from linking a user's PID_Us and $Account$s at different RPs, and therefore prevents RP-based identity linkage.

B. UPPRESSO Overview

The UPPRESSO system has four procedures, namely system initialization, RP initial registration, user registration, and SSO login.

System Initialization. System initialization is conducted once by the IdP to establish the entire system. In particular, the IdP generates a large prime p, a prime factor q of $p-1$ and a generator g of order q as the parameters of the discrete logarithm problem. The IdP also generates one key pair (SK, PK) to sign identity proofs and RP certificates. The lengths of p, q and (SK, PK) should satisfy the required security strength. Then, the IdP keeps SK secret, while announcing p, q, g and PK are public parameters.

RP Initial Registration. RP initial registration is launched by each RP to obtain the necessary configurations from the IdP, including a unique identifier ID_{RP} and its corresponding RP certificate $Cert_{RP}$. Each RP launches this procedure only once. In particular, an RP registers itself at the IdP and requests ID_{RP} and $Cert_{RP}$ as follows:

- The RP sends a registration request to the IdP, including the RP endpoint (e.g., URL) to receive identity proofs.
- The IdP chooses a unique random number r ($1 < r < q$), calculates $ID_{RP} = g^r \bmod p$, signs $[ID_{RP}, Endpoint_{RP}, *]$ using SK, where $*$ denotes the supplementary information such as the RP's common name, and returns $Cert_{RP} = [ID_{RP}, Endpoint_{RP}, *]_{SK}$ to the RP, where $[\cdot]_{SK}$ means the message is signed using SK.
- The RP verifies $Cert_{RP}$ using PK and accepts ID_{RP} and $Cert_{RP}$ if they are valid.

Note that, ID_{RP} cannot be chosen by the RP. It must be chosen by the IdP with r unknown to the RP.

User registration. UPPRESSO takes the same user registration process as other SSO systems to set up a unique user identifier ID_U and the corresponding user credential. User registration is launched only once by each user. ID_U can be chosen by either the user or the IdP, as long as it is unique for each user.

SSO Login. Finally, an SSO login procedure will be launched when a user attempts to log into an RP. It is designed on top of the identifier-transformation functions. As shown in Figure 3, the SSO login consists of four phases, namely, RP identifier transformation, PID_{RP} registration, identity proof generation and $Account$ calculation. In RP identifier transformation, the user and the RP negotiate $PID_{RP} = {ID_{RP}}^{N_U N_{RP}} \bmod p$. Next, the user registers PID_{RP} at the IdP. It is worth noting that this step has to be conducted by the user but not the RP. Otherwise, the IdP can associate PID_{RP} and ID_{RP}. To register PID_{RP}, the user needs to create a new endpoint and submit it with PID_{RP} to the IdP. Then, the RP requests the identity proof, and the IdP calculates $PID_U = {PID_{RP}}^{ID_U} \bmod p$ and signs the identity proof. Finally, in $Account$ calculation, the RP derives $Account = {PID_U}^{(N_U N_{RP})^{-1} \bmod q} \bmod p$ after verifying the identity proof and allows the user to log in under $Account$.

Moreover, as the IdP is unaware of the visited RP and also the RP's endpoint to receive the identity proof, this endpoint shall be queried by the user from the trusted IdP indirectly to ensure confidentiality; otherwise, an incorrect endpoint leaks the identity proofs. In UPPRESSO this is implemented as an RP certificate signed by the IdP, which is composed of ID_{RP}, the RP's endpoint and other supplementary information. Then, the user determines the correct endpoint by itself, while in commonly-used OIDC systems, the endpoint is configured by the IdP.

C. SSO Login Flow of UPPRESSO

We illustrate the steps of the SSO login protocol of UPPRESSO in Figure 3, and describe the detailed processes as follows.

RP Identifier Transformation. In this phase, the user and the RP cooperate to generate PID_{RP} as follows:

1.1 The user sends a login request to trigger the negotiation of PID_{RP}.

1.2 The RP chooses a random number N_{RP} ($1 < N_{RP} < q$), calculates $Y_{RP} = {ID_{RP}}^{N_{RP}} \bmod p$, and sends Y_{RP} with $Cert_{RP}$ to the user.

1.3 The user verifies $Cert_{RP}$, extracts ID_{RP} from the valid $Cert_{RP}$, chooses a random number N_U ($1 < N_U < q$) to calculate $PID_{RP} = {Y_{RP}}^{N_U} \bmod p$, and sends N_U to the RP.

1.4 The RP verifies $N_U \neq 0 \bmod q$, calculates PID_{RP} with N_U and Y_{RP}, derives the trapdoor $T = (N_U N_{RP})^{-1} \bmod q$; and acknowledges the negotiation by responding with N_{RP}.

1.5 The user verifies that $N_{RP} \neq 0 \bmod q$ and $Y_{RP} = {ID_{RP}}^{N_{RP}} \bmod p$.

The user halts the negotiation, if $Cert_{RP}$ is invalid, $N_{RP} = 0 \bmod q$, or $Y_{RP} \neq {ID_{RP}}^{N_{RP}} \bmod p$. The verification of Y_{RP} and N_{RP} ensures the order of Y_{RP} (and also PID_{RP}) is q, and prevents a malicious RP from choosing an arbitrary Y_{RP} (then PID_{RP}) of order less than q, which makes it less difficult for the RP to derive ID_U from PID_U.

$\mathbf{PID_{RP}}$ Registration. The user registers PID_{RP} at the IdP.

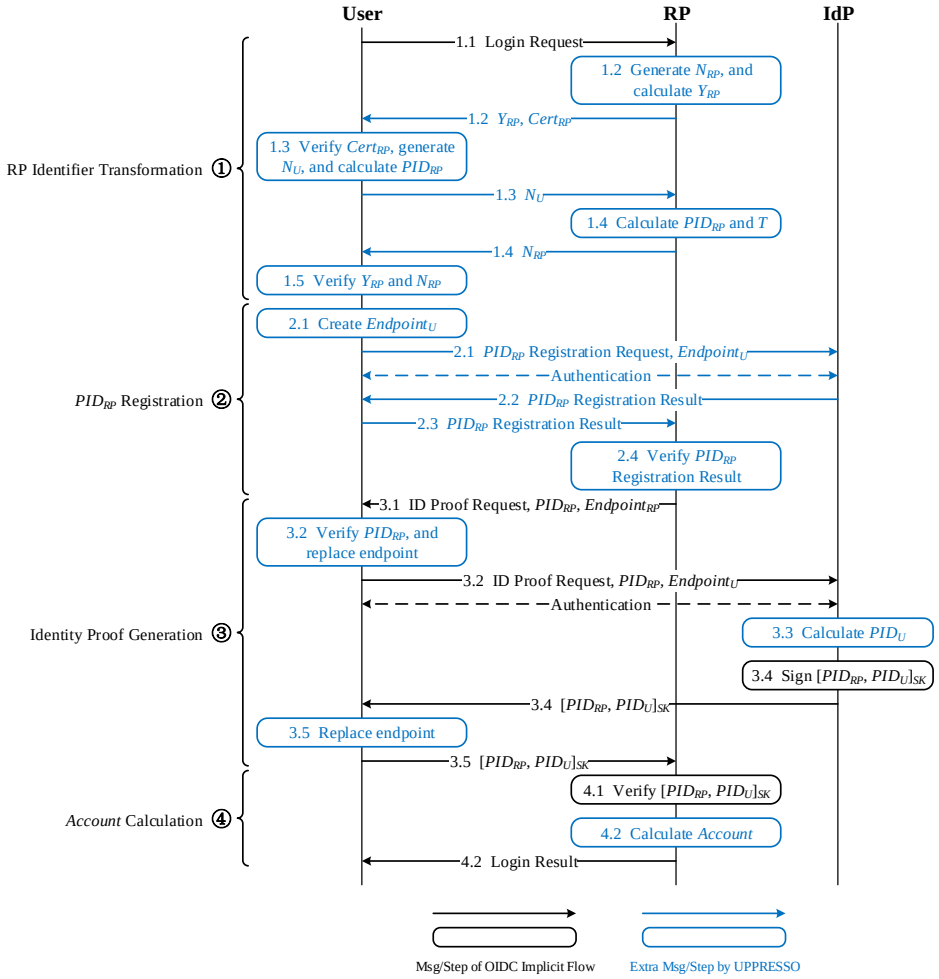

Fig. 3: The flow of a user login in UPPRESSO.

2.1 The user creates a one-time endpoint to hide the RP's endpoint from the IdP, and sends the PID_{RP} registration request $[PID_{RP}, Hash(N_{RP}, N_U), Endpoint_U]$ to the IdP.

2.2 The IdP authenticates the user if she has not been authenticated yet. The IdP verifies that PID_{RP} is unique among unexpired PID_{RP}s, and then signs the response $[PID_{RP}, Hash(N_{RP}, N_U), Validity]_{SK}$, where $Validity$ is the validity period. The IdP returns the signed response to the user.

2.3 The user forwards the registration result to the RP.

2.4 The RP verifies the IdP's signature, and accepts it only if PID_{RP} and $Hash(N_{RP}, N_U)$ match those in the negotiation and it is in the validity period.

$Hash(N_{RP}, N_U)$ is attached as the nonce to avoid the registration result is accepted by two or more RPs, which have different ID_{RP}s but generate a same PID_{RP}. The IdP ensures PID_{RP} is unique among unexpired ones; otherwise, one identity proof for one PID_{RP} might be accepted by other RPs. More details are analyzed in Section **??**.

ID Proof Generation. In this phase, the user login continues and the IdP signs the identity proof.

3.1 The RP uses PID_{RP} and $Endpoint_{RP}$ to construct an identity proof request for a set of user's attributes.

3.2 The user first confirms the scope of the requested attributes and verifies PID_{RP} with the negotiated one. The user replaces the endpoint with the registered one-time $Endpoint_U$, and sends the modified identity proof request to the IdP.

3.3 The IdP verifies whether PID_{RP} and $Endpoint_U$ have been registered and unexpired, and calculates $PID_U = PID_{RP}{}^{ID_U} \bmod p$ for the authenticated user.

3.4 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, $Attr$ contains the requested attributes. Then, the IdP sends the identity proof to the one-time endpoint at the user.

3.5 The user extracts the RP endpoint in $Cert_{RP}$, and forwards the identity proof to the RP through this endpoint.

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.

Account calculation. Finally, RP derives the user's $Account$ and completes the user login as follows.

4.1 The RP verifies the identity proof, including the signature, validity period, and the consistency between PID_{RP} and the negotiated one. If any fails, the RP rejects this login.

4.2 The RP extracts PID_U, calculates $Accout = {PID_U}^T \bmod p$, and allows the user to log in.

D. Compatibility with OIDC

As described above, the SSO login protocol of UPPRESSO follows a same logic flow of OIDC login protocol with small modifications to transform the identifiers. Next, we will discuss these necessary modifications and demonstrate its compatibility with OIDC. This indicates that it can be easily integrated with other commonly adopted SSO systems.

First, UPPRESSO does not introduce any new role nor change the security assumptions for each role. Moreover, among the four phases of its SSO login flow, only the RP identifier transformation phase is newly introduced by UPPRESSO, while the other three (PID_{RP} registration, identity proof generation and $Account$ calculation) adopt similar communication pattern as OIDC flows.

In particular, the PID_{RP} registration phase can be viewed as a variant of the RP dynamic registration flow of OIDC [**?**], where an entity registers its identity and endpoint at the IdP. Different from OIDC, in UPPRESSO, this process can be launched by any authenticated user who obtains an RP identifier, the registration response includes a signature by the IdP, and the registration will become invalid after a validity period. However, these changes only require small modifications to the RP dynamic registration flow of OIDC.

The identity proof generation and $Account$ calculation phases adopt the same steps and functions as the implicit protocol flow of OIDC but calling a few different parameters. First, in identity proof generation, PID_U transformed from ID_U is used to replace ID_U, which is directly supported by OIDC, similar as in PPID approaches that also convert ID_U into PID_U. The calculation of $Account$ from PID_U can be viewed as a customized step by the RP to derive its user account after the implicit protocol flow of OIDC ends. Another modification is the replacement of endpoint by the user for transmitting the identity proof to the RP. In fact, this message forwarding is common when an application-layer network proxy is deployed. So, the identity proof generation and $Account$ calculation phases of UPPRESSO can be viewed as a particular but compatible implementation of the implicit protocol flow of OIDC.

It is worth noting that the identity proof generation and $Account$ calculation phases of UPPRESSO can be also implemented as the authorization code flow of OIDC with small modifications, which will be discussed in Section VIII.

VI. Web Model

Our formal analysis of UPPRESSO is based on the Dolev-Yao style web model [**?**], which has been widely used in formal analysis of SSO protocol, e.g., OAuth 2.0 [**?**] and OIDC [**?**]. To make the description cleaner, we foucus on our modification on OIDC, and assume DNS and HTTPS are secure, which has already been anaylzed in [**?**].

The main entities in the model are $atomic\ processes$, which represent the essential nodes in the web systems, such as browsers, web servers and attackers. The atomic processes communicate with each other through the $events$ containing the receiver atomic process's address (IP), the sender atomic process process's address (IP) and the transmitted $message$. Moreover, there are also dependent $scripting\ processes$ which runs on the client-side environment relying on the browsers such as JavaScript. The scripting provides the server defined function to the browser. The web system mainly consists of the set of atomic processes and scripting processes. The operation of a system is described as that the system converts its states via step of runs. The state of web system is called $configuraton$ which consists of all the states of the atomic processes in the system and all the event can be accepted by the processes.

A. Communication Description

Here we give a brief presentation of generic Dolev-Yao-style communication model proposed by [**?**] on which our web model is based.

A $signature\Sigma$ consists of a finite set of function symbols, such as `encrypt`, `decrypt`, and `pair`, each with an arity. A function symbol with arity 0 (with no arguments) is a constant symbol. The set of *terms* is defined over a signature Σ, an infinite set of names, and an infinite set of variables.

Messages are defined as formal terms without variables (called ground terms). The signature Σ for the messages in the model is considered containing constants (such as ASCII strings and nonce), sequence symbols (such as n-ary sequences $\langle\rangle$, $\langle.\rangle$, $\langle.,.\rangle$ etc.) and further function symbols (such as encryption/decryption and digital signatures). An HTTP request is a common message in the web model, containing a type `HTTPReq`, a nonce n, a method `GET` or `POST`, a domain , a path, URL parameters, request headers, and the body over the Σ in the sequence symbol formate. Here is an example for an HTTP GET request for the domain `exa.com/path?para` $= 1$ with the headers and body empty.

$$m := \langle \texttt{HTTPReq}, n, \texttt{GET}, exa.com, /path, \langle\langle para, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

Events are the basic communication elements in the model. An event is the term in the formate $\langle a, f, m\rangle$, where the a and f represent the address of sender and receiver, and m is the message transmitted.

Atomic Processes. An $atomic\ Dolev-Yao(DY)\ process$ is constructed as the tuple $p = (I^p, Z^p, R^p, s_0^p)$ representing the single node in the web model, such as the server and browser. I^p is the set of addresses a process listens to, and Z^p is the set of states (terms) which describes the process. R^p is the mapping between the pairs $\langle s, e\rangle$ and $\langle s', e'\rangle$ where $s, s' \in Z^p$

It's worth noting that for one process in a state only a finite set of events can be accepted by the process as the state and event are defined as the input of R^p.

Scripting Processes. The web model also contains the scripting process representing the client-side script loaded by browser such as JavaScript code. However, the $scripting\ process$ must rely on an $atom\ process$ such as browser and provide the relation R witch is called by this $atomic\ process$.

Equational theory is defined as usual in Dolev-Yao models which introduces the symbol $\equiv$ representing the congruence relation on terms. For instance, $dec(enc(m,k),k) \equiv m$

B. Web System

The web system contains a set of processes (including atomic processes and scripting processes) and represents the web infrastructure. A web system is defined as a tuple ($\mathcal{W}$, $\mathcal{S}$, `script`, E^0). $\mathcal{W}$ is the set of atomic processes containing honest processes and malicious processes, $\mathcal{S}$ is the set of scripting processes including honest scripts and malicious scripts, `script` is the set of concrete script code related with specific scripting process in $\mathcal{S}$, and E^0 is the set of events which could be accepted by the processes in $\mathcal{W}$.

Configuration. We firstly define the set of states S of a system, consists of all the current states of processes in $\mathcal{W}$. And the set of events E, for each event $e \in E$, there is always a state $s \in S$, e and s can be accepted by one of the processes as the input. A $configuration$ of the system is defined as the tuple (S, E, N) where N is the mentioned sequence of unused nonces.

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

C. Model Of UPPRESSO

The UPPRESSO model is a web system which is defined as

$$\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \texttt{script}, E^0),$$

$\mathcal{W}$ is the finite set of atom processes in UPPRESSO system including a single IdP server process, multiple honest RP server processes , the browsers representing the users, and the attacker processes. We assume that all the honest RPs are implemented following the same rule so that the process are considered consistent besides of the addresses they listen to. The browsers controlled by user are considered honest. That is, the browser controlled by attackers can behave as an independent atomic process. $\mathcal{S}$ is the finite set of scripting processes consists of $script\_rp$, $script\_idp$ and $script\_attacker$. The $script\_rp$ and $script\_idp$ are downloaded from honest RP and IdP processes and the $script\_attacker$ is downloaded from attacker process considered existing in all browser processes.

We now give a brief description about UPPRESSO model.

- The browser is responsible to send HTTP request, receive HTTP response, handle user behaviour and transmit message between scripting process. As the browser is honest, we only focus on the scripting process running on the browser. The detailed model of browser is shown in [**?**]
- IdP server process (defined as p^i) only accepts the events whose messages are HTTP request and the $path \in /script$, $/dynamicRegistration$, $/login$, $/loginInfo$, $/authorize$. The function of each path is shown in Section V. All the events are accepted by p^i in any state but the output may be different. The detailed R^i is shown in *.
- RP server process (defined as p^r) only accepts the events whose messages are HTTP request and the $path \in \{$ $/script$, $/login$, $/startNegotiation$, $/registrationResult$, $/uploadToken$ $\}$. The function of each path is shown in Section V. The event with $path \in \{$ $/script$, $/login$, $/startNegotiation$ $\}$ are accepted in any state. However, the event with $path \equiv /registrationResult$ is accepted only when the state s is the output for event $path \equiv /startNegotiation$. In the same way the following accepted events must be arranged as $path$ in the sequence $/registrationResult$, $/uploadToken$.
- IdP and RP scripting process accepts the events in the formate as HTTP response and postMessage. The details about accepted events are shown in **??**.

D. Security Of UPPRESSO

As we assume that the HTTP requests and responses are well protected by TLS, and the postMeassage are securely implemented in browser, therefore, web attackers are not considered. In this section, we are going to prove the following theorem,

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

Firstly, an SSO system is considered secure **iff** only the legitimate user can always log into an honest RP under her unique account. Based on the model of UPPRESSO, we found that an attacker can visit an honest RP as the honest user only when the attacker own the cookie which is bound to the honest user by RP. Therefore, the definition of a secure UPPRESSO system is,

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.

Therefore, to prove theorem 1, we are going to prove that an authenticate cookie c is unknown to attacker a. The proof can be separated as two parts, initially a does not know any authenticated cookie, and the following requirements must be met.

- If c is the authenticated cookie owned by u, c cannot be obtained by a;.
- If c is an unauthenticated cookie owned by a, c cannot be authenticated by r for u.

Proof Outline. Here we introduce the lemmas briefly to prove that $\mathcal{UWS}$ follows the requirements by Definition 1 so that

$\mathcal{UWS}$ is secure. And the detailed proofs to these lemmas are in **??**.

Lemma 1. The cookie owned by honest user will not be leaked to any attacker.

Proof: That is, due to the Same-Origin policy, the honest browser will not leak the cookies to any attacker. And based on the UPPRESSO model, it is to prove that RP server and RP script will not send any cookies to other processes either. Therefore, the attackers cannot obtain the u's authenticated cookie. ■

Based on the model of UPPRESSO about RP server process, the procedure of the cookie being authenticated is described as follows.

Definition 2. In $\mathcal{UWS}$, the cookie c is to be set authenticated for user u only when RP r receives a valid u's identity proof from the owner of c.

Then we are going to prove that $\mathcal{UWS}$ follows the requirements that the cookie of the attacker cannot be set authenticated.

Here we propose the lemmas

Lemma 2. Attackers cannot obtain the user u's password in $\mathcal{UWS}$.

Lemma 3. Attackers cannot forge the IdP issued proofs in $\mathcal{UWS}$.

Proof: Lemma 2 can be easily proved because the password is only sent by honest IdP scripting process to IdP server. Lemma 3 is proved as the IdP issued proofs are well signed and verified. Therefore, the following lemma can be proved base on Lemma 2 and Lemma 3. ■

Lemma 4. Attackers cannot obtain the u's valid identity proof in $\mathcal{UWS}$.

Proof: We now give a brief proof about Lemma 4. As the attacker attempts to obtain a valid identity proof, he must receive the proof from one of following processes, IdP server process, RP server process, IdP scripting process and RP scripting process. That is, according to the model we find the honest RP scripting process only send identity proof to honest RP server and RP server will not send the proof to any process. It can be proved that only the process who holds u's password can obtain the u's identity proof from IdP server. As the attacker does not know u's password so that he cannot obtain the identity proof from IdP server. To prove that attacker cannot obtain the identity from IdP scripting process is a little complicated so that we here only give a straightforward conclusion. That is when the honest user u sends the identity proof from the IdP scripting process, the receiver is restricted by the RP Certification $cert_r$. And the identity proof is valid in honest RP r only if the $cert_r$ belongs to r (the full proof is in **??**). ■

Therefore, $\mathcal{UWS}$ satisfies the requirements in Definition **??**, such that Theorem 1 is proved.

E. Privacy Of UPPRESSO

Firstly we introduce the definition in [] about static equivalence.

Definition 3. Two messages t_1 and t_2 are statically equivalent, written $t_1 \approx t_2$, if and only if, for all terms such as $M(x)$ and $N(x)$ which only contain one variable x without nonces, it is true that $M(t_1) \equiv N(t_1)$ **iff** $M(t_2) \equiv N(t_2)$. For instance, there are the messages m and m', symmetric key k, such that $enc(m,k) \approx enc(m',k)$ is always true to the attackers without the k.

Here we give the new definitions

Definition 4. For a large prime p (2048-bit length) and $p-1$'s prime factor q (256-bit length), there are two constants g_1, g_2 as the generators of p and the constants n_1, n_2 (n_1, n_2 $< q$). We define the function symbol $modpow(a,b,p) = a^b \mod p$, there are $modpow(g_1,n_1,p) \approx modpow(g_2,n_2,p)$ and $modpow(g_1,n_1,p) \approx modpow(g_1,n_2,p)$ always true due to the discrete logarithm problem as the n_1 and n_2 are unknown.

Definition 5. Equivalence of HTTP requests. There are messages m_1 and m_2, we say that $m_1 \approx m_2$ **iff** the following conditions are met,

- If m_1 and m_2 are HTTPs requests, they are equivalent to the observers besides of the receiver.
- If m_1 and m_2 are HTTPs requests, they are equivalent for the receiver **iff** the value of the Host,Path,Origin and Referer headers in both requests are same, as well as the value of the Parameters and Body are statically equivalent.
- If m_1 and m_2 are HTTP requests, they are equivalent to all the observers as the equivalent HTTPS requests to receivers.

Definition 6. Equivalence of events. There are events $e_1 := \langle a_1, f_1, m_1 \rangle$ and $e_2 := \langle a_2, f_2, m_2 \rangle$, we say that $e_1 \approx e_2$ **iff**

- $a_1 \equiv a_2$ or a_1 and a_2 belong to random addresses.
- $f_1 \equiv f_2$ or f_1 and f_2 belong to random addresses.
- m_1 and m_2 are equivalent.

Then we are going to prove the following theorem

Theorem 2. *Let $\mathcal{UWS}$ be a UPPRESSO web system, then $\mathcal{UWS}$ is IdP-Privacy and RP-Privacy.*

The definitions about IdP-Privacy and RP-Privacy are designed as follows.

Definition 7. IdP-Privacy Let $\mathcal{UWS}$ be a UPPRESSO web system, there are honest RPs $r_1, r_2 \in \mathcal{W}$, IdP $i \in \mathcal{W}$ and the honest user u, then $\mathcal{UWS}$ is IdP-Privacy **iff** for every event e_1 received by i during the u logging in to r_1, there is always an event e_2 for the u logging in to r_2, and e_1 and e_2 are equivalent.

Proof: Here we only give a brief proof that $\mathcal{UWS}$ meets the conditions defined in Definition 7. Firstly, it is assumed that

the HTTPs transmissions well implemented such that all the events to IdP are regarded as equivalent to web attackers. As we consider IdP server is honest but curious, i can only hold the events to IdP server process and does not attempt to steal parameters from other processes or set any illegal parameters in the system.

Here we only focus on the same user's multiple requests to the IdP. IdP server only accepts the events whose messages are HTTP request and the $path \in /script$, $/dynamicRegistration$, $/login$, $/loginInfo$, $/authorize$. All the path will be visited in each login procedure. It can be easily found that the visits to $/script$ and $/loginInfo$ carrying no parameters and bodies so that the events must be equivalent. The visits to $/login$ only carry u's username and password so that the events are equivalent. Moreover ,the visits to $/dynamicRegistration$ and $/authorize$ carry the PID_{RP}s and $endpoint$s where PID_{RP}s are statically equivalent because of Definition 4 and $endpoint$s are unrelated random constants . Therefore, $\mathcal{UWS}$ meets the conditions defined in Definition 1, so that theorem 1 is proved. ■

Definition 8. RP-Privacy Let $\mathcal{UWS}$ be a UPPRESSO web system, there are honest RPs $r_1, r_2 \in \mathcal{W}$ and the honest users u_1 and u_2, then $\mathcal{UWS}$ is RP-Privacy **iff** event through r_1 and r_2 share their states

- for every event e_1 received by r_2 during the u_1 logging in to r_2, there is always an event e_2 for the u_2 logging in to r_2, and e_1 and e_2 are equivalent to r_1.
- for every events received by r_2, the event cannot be straightforward linked to the existing user's attributes at r_1.

RP server process only accepts the events whose messages are HTTP request and the $path \in \{ /script, /login, /startNegotiation, /registrationResult, /uploadToken \}$. As the RPs may behave malicious so that the events received by RP scripting process should also be considered. However, all of the messages received by RP scripting process are transmitted to RP server. Therefore, we only need to focus on the events received by RP server.

Firstly, we assume that all the parameters are set legally. We give the brief proof. The events visiting to $/script$ and $/login$ carry no parameters and bodies so that the events must be equivalent. The visits to $/startNegotiation$ only carry the nonce so that the events are equivalent. The visits to $/registrationResult$ carry the IdP signed registration result, however, the contents in the result contains the PID_{RP}, N_U and $endpint$. The contents are all random constants (PID_{RP} is regarded as same as N_U) so that the events are equivalent. The visits to $/uploadToken$ includes the identity proof containing the PID_{RP}, PID_U. According to Definition 4, the PID_Us are statically equivalent to r_1. Moreover, with the r_2 shared state, $Account_{r_1}$s are known to r_1. However, r_1 is unable to transform $Account_{r_1}$s into the users' account $Account_{r_2}$ at r_2 so that the events cannot be linked to the existing user. Therefore, the requirements of Definition 8 are met.

However, as the RPs are considered maybe malicious, such that they will attempt to steal the data from other process or set the malicious parameters during the login procedure. That is, according to Definition 4, the PID_U the $Accounts$ must be equivalent to the attacker as long as the attacker does not know the ID_U. Therefore the attacker may attempt to steal the ID_U from UPPRESSO system. But it is easy to be found that IdP will not send the plain ID_U to any process so that RPs cannot obtain the ID_U. Another way is that RPs may attempt to treat the $Account$ or PID_U to be generated insecurely, but we are going to prove it is impossible.

- RP may lead the login using the forged ID_{RP} or PID_{RP} so that PID_Us and $Accounts$ are no more equivalent. However, ID_{RP} are provided by the $Cert$, protected by the IdP's signature and verified by IdP script. PID_{RP} is generated by the ID_{RP} and the honest user generated nonce. Therefore, it is impossible to lead the honest user to use the illegal ID_{RP} and PID_{RP}.
- RP may also lead the same user to upload the identity proof with same PID_U or $Account$ so that the system is not RP-Privacy according to Definition 8. However, the PID_U is generated with the user's nonce N_U so that it is not controlled by the RP. $Account$ is generated as the form $ID_{RP}^{ID_U} \mod p$, while RPs may lead the user to use the same ID_{RP} to generate identity proof. However, the ID_{RP} is bound with $Cert$ which is verified by the user and it is easy for user to find out the login RP does not coincide the RP name shown on her browser.

Therefore, we consider that $\mathcal{W}$ meet all the requests defined in Definition 8 so that theorem 2 is proved.

VII. Implementation and Performance 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.

A. Implementation

We adopt SHA-256 for digest generation, and RSA-2048 for signature generation. We randomly choose a 2048-bit prime as p, a 256-bit prime as q, and a q-order generator as g. N_U, N_{RP} and ID_U are 256-bit random numbers. Then, the discrete logarithm problem provides equivalent security strength (i.e., 112 bits) as RSA-2048 [**?**]. 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, 26 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 Chrome extension with about 330 lines of JavaScript code, to provide the functions in Steps 1.3, 1.5, 2.1, 3.2 and 3.5. The cryptographic computations, e.g., $Cert_{RP}$ verification and PID_{RP} negotiation, are implemented based on jsrsasign [**?**], an efficient JavaScript cryptographic library. This chrome extension requires permissions *chrome.tabs* and *chrome.windows* to obtain the RP's URL from the browser's tab, and *chrome.webRequest* to intercept, block, modify requests to the IdP or RP [**?**]. Here, the cross-origin HTTPS requests sent by this chrome extension to the RP and IdP, will be blocked by Chrome due to the default same-origin security policy. To avoid this block, UPPRESSO modifies the IdP and RP, and sets `chrome-extension://chrome-id` (`chrome-id` is uniquely assigned by Google) in `Access-Control-Allow-Origin` header of the IdP's and RP's responses.

We provide a Java SDK for RPs to integrate UPPRESSO. The SDK provides 2 functions to encapsulate RP's processings: one for RP identifier transformation, PID_{RP} registration and identity proof request generation; while the other for identity proof verification and $Account$ calculation. The SDK is implemented based on the Spring Boot framework with about 1100 lines code, and cryptographic computations are implemented based on Spring Security library. An RP only needs to invoke these two functions for the integration.

B. Performance Evaluation

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

Setting. We compare UPPRESSO with MITREid Connect [**?**] 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 the three schemes provide the same function, i.e., extracting the user's account from the identity proof. We have measured the time for a user's login at an RP, and calculated the average values of 1000 measurements. For better analysis, we divide a login into 4 phases according to the lifecycle of identity proof: **Identity proof requesting** (Steps 1.1-3.2 in Figure 3), the RP (and user) constructing and transmitting the request to IdP; **Identity proof generation** (Steps 3.3 and 3.4 in Figure 3), the IdP generating identity proof (no user authentication); **Identity proof extraction** (Steps 3.4 and 3.5 in Figure 3), the RP server extracts the identity proof from the IdP; and **Identity proof verification** (Steps 4.1 and 4.2 in Figure 3), the RP verifying and parsing the identity proof.

Results. The evaluation results are provided in Figure 4. The overall processing times are 113 ms, 308 ms and 254 ms for MITREid Connect, SPRESSO and UPPRESSO, respectively. The details are as follows.

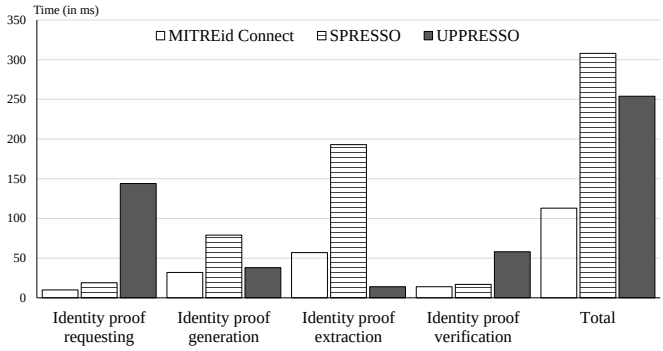

Fig. 4: The Evaluation.

In the requesting, UPPRESSO requires both the user and RP to perform 2 modular exponentiations for RP identifier transformation and complete PID_{RP} registration at the IdP, which totally need 144 ms; SPRESSO needs 19 ms for the RP to obtain IdP's public key and encrypt its domain; while MITREid Connect only needs 10 ms.

In the generation, UPPRESSO needs an extra 6 ms for 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. As the processings in SPRESSO and MITREid Connect are the same, the processing time in SPRESSO may be reduced to 32 ms. And, then the overall time in SPRESSO will be 269 ms, still larger than 254 ms in UPPRESSO.

In the identity proof extraction, UPPRESSO only needs 14 ms where the Chrome extension relays the identity proof to the RP server directly. 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 57 ms. SPRESSO needs the longest time (193 ms) due to a complicated processing at the user's browser, which needs the browser to obtain identity proofs from the IdP, download the JavaScript program from a trusted entity (forwarder), execute the program to decrypt RP's endpoint, send identity proofs to this endpoint (an RP's web page) who finally transmits the proof to RP server. In the evaluation, the forwarder and IdP are deployed in one machine, which doesn't introduce performance degradation based on the observation.

In the verification, UPPRESSO needs an extra calculation for $Account$, which then requires 58 ms, compared to 14 ms in MITREid Connect and 17 ms in SPRESSO.

VIII. Discussions and Future work

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

Scalability. The adversary cannot exhaust ID_{RP} and PID_{RP}. For ID_{RP}, it is generated only in RP's initial registration. For PID_{RP}, in practice, we only need to ensure all PID_{RP}s are different among the unexpired identity proof (the number denoted as n). We assume that IdP doesn't perform the uniqueness check, and then calculate the probability that at

least two PID_{RP}s are equal in these n ones. The probability is $1-\prod_{i=0}^{n-1}(1-i/q)$ which increases with n. For an IdP with throughput $2*10^8$ req/s, when the validity period of the identity proof (PID_{RP}) is set as 5 minutes, n is less than 2^{36}, then the probability is less than 2^{-183} for 256-bit q. Moreover, as this probability is negligible, the uniqueness check of PID_{RP}, i.e., the PID_{RP} registration, could be removed in the SSO login process, and this optimization can be adopted when this negligible probability is acceptable by the users and RPs.

Security against DoS attack. The adversary may attempt to perform DoS attack on the IdP and RP. For example, the adversary may act as a user to invoke the PID_{RP} registration (Step 2.1) and identity proof generation (Step 3.2) at the IdP, which requires the IdP to perform two signature generations and one modular exponentiation. However, as the user has already been authenticated at the IdP, the IdP could identify the malicious users based on audit, in addition to the existing DoS mitigation schemes. The adversary may act as a user requesting to log into an RP, and make the RP perform two modular exponentiations. The RP could previously calculated a set of Y_{RP}s to mitigate this attack.

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.

Platform independent. Our current implementation only requires the user to install a Chrome extension and doesn't need to store any persistent data at the user's machine. Moreover, the implementation could be further extended to remove the Chrome extension, whose JavaScript program is then fetched from the honest IdP. The processing is similar as SPRESSO. That is, 1) the RP's window (window A) opens a new iframe (window B) to visit the RP's web page, while the RP's web page redirects window B to the IdP; 2) window B downloads the JavaScript program from the IdP and performs the processing in Steps 1.3, 1.5, 2.1, 3.2 and 3.5; 3) then postMessages are adopted to exchange messages between window A and B for Steps 1.2, 1.3, 1.4, 2.3, 3.1 and 3.5. The opener handle of window B is preserved (i.e., window A) for the postMessage, as window A opens window B with a web page from the RP; and window B is redirected to the IdP with *noreferrer* attribute set, to prevent the browser from sending RP's URL in the Referrer header to the IdP.

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 [**?**].

IX. 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 and etc. And, plenty of works have been conducted on privacy protection and security analysis for SSO systems.

A. Privacy protection 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 and etc., were adopted to design anonymous SSO schemes [**?**], [**?**]. Anonymous SSO schemes are designed for the anonymous services, and not applicable to common services which need user identification.

B. Security analysis of SSO systems.

Formal analysis on SSO standards. The SSO standards (e.g., SAML, OAuth and OIDC) have been formally analyzed. Fett et al. [**?**], [**?**] have conducted the 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, 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 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.

Single sign-off. In SSO systems, once a user's IdP account is compromised, the adversary could hijack all her RPs' accounts. A backwards-compatible extension, named single sign-off, is proposed for OIDC. The single sign-off allows the user to revoke all her identity proofs and notify all RPs to freeze her accounts [**?**]. The single sign-off could also be achieved in

UPPRESSO, where the user needs to revoke the identity proofs at all RPs, as the IdP doesn't know which RPs the user visits.

Analysis on 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, bidirectional authenticated secure channel was proposed to improve the confidentiality and integrity of identity proof [**?**]. The 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 RP designation of identity proof.

Analysis on mobile SSO systems. In mobile SSO systems, the IdP App, IdP-provided SDK (e.g., an encapsulated WebView) or system browser are adopted to redirect identity proof from IdP App to RP App. However, none of them was trusted to ensure that the identity proof could be only sent to the designated RP [**?**], [**?**], as WebView and system browser cannot authenticate RP App while the IdP App may be repackaged. Moreover, the SSO protocols needed to be modified to provide SSO services for mobile Apps, however these modifications were not well understood by RP developers [**?**], [**?**]. The top Android applications have been analyzed [**?**], [**?**], [**?**], [**?**], [**?**], and vulnerabilities were found to break the confidentiality [**?**], [**?**], [**?**], [**?**], [**?**], integrity [**?**], [**?**], and RP designation [**?**], [**?**] of identity proof.

X. CONCLUSION

In this paper, we propose UPPRESSO, an unlinkable privacy-preserving single sign-on system, which protects a user's login activities at different RPs against both curious IdP and collusive RPs. To the best of our knowledge, UPPRESSO is the first approach that defend against both IdP-based login tracing and RP-based identity linkage privacy threats at the same time. To achieve these goals, we convert the privacy problem in SSO services into an identifier-transformation problem and design three transformation functions based on the discrete logarithm problem, where $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ prevents curious IdP from knowing the identity of the RP, $\mathcal{F}_{ID_U \mapsto PID_U}$ prevents collusive RPs from linking a user based on her identifier, and $\mathcal{F}_{PID_U \mapsto Account}$ allows each RP to derive an identical account for a user in her multiple logins. The three functions could be integrated with existing SSO protocols, such as OIDC, to enhance the protection of user privacy, without breaking any security guarantee of SSO. Moreover, the evaluation on the prototype of UPPRESSO demonstrates that it supports an efficient SSO service, where a single login takes only 254 ms on average.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

APPENDIX A
WEB MODEL

A. Data Formate

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

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

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

, and an HTTP response message is the term of the form

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

The details are dined as follows:

- `HTTPReq` and `HTTPResp` are the type of messages.
- $nonce$ is the constant nonce mapping the response with the specific request.
- $method$ is the HTTP method, 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 $parameters$ HTTP request sent to 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. A 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 same as in HTTP message.

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 defined as 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 the for the receiver, $Content$ is the message transmitted and Origin is restricts the receiver's origin.

XMLHTTPREQUEST. XMLHTTPRequest is the HTTP message transmitted by scripts in the browser. That is the XMLHTTPRequest is converted with 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 $\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle$. We assume there is an HTTP request $r := \langle \texttt{HTTPReq}, n, \texttt{GET}, example.com, /path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$, here we define the operation on the r. That is the elements in r can be accessed in the form $r.name$, such that $r.method \equiv \texttt{GET}$, $r.path \equiv /path$ and $r.body \equiv \langle\rangle$.
- **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 $\star$ as the pattern, such as $\langle a, b, \star\rangle$. That is the pattern matches any terms which is in the same form of the pattern but replacing the $\star$ with other terms. For instance, $\langle a, b, \star\rangle$ matches $\langle a, b, c\rangle$.

B. Browser Model

As we consider that the browsers are honest in UPPRESSO model, therefore, we only focus on 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 widow is created in which document, 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 $nonce$ is to locate the document. $Location$ is the URL where the document is loaded. $Referrer$ is same as the Referer header defined in HTTP standard. $script$ is the scripting process downloaded from each servers. $scriptstate$ is define by the script, different in each scripts. $scriptinputs$ is the message transmitted into the scripting process. $subwindows$ 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 $scriptinputs$ are defined as standardized forms, for example postMessage is one of the forms of $scriptinputs$. $Cookies$ is the set of cookies 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 $\langle \texttt{IFRAME}, URL, WindowNonce \rangle$ asks the browser to create this document's subwindow and it visits the server with the URL.

C. Model Of UPPRESSO

In this section, we will introduce the model of processes in UPPRESSO system, containing 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 states, and the content of new output events and states.

D. IdP Server Process

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

- ID is the identifier of IdP.
- p is the large prime mentioned before.
- $SignKey$ 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 identify the session and sessions store the browser uploaded message.
- $users$ is the set of user information. And each user informations contains the $username, password, uid$ and other user attributes.
- RPs is the set of RP information which consists of ID of RP (PID_{RP}), $Endpoints$ 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 uid.
- `ListOfPID()` is the set of IDs of registered RP.
- `EndpointsOfRP(r)` is the set of endpoints registered by the RP with ID r.
- `ModPow(a, b, c)` is the result of $a^b \mod c$.
- `CurrentTime()` is the system current time.

The relation of IdP process R^i is shown as Algorithm 1.

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**
let $cookie := headers[Cookie]$
let $session := s'.sessions[cookie]$
let $username := session[username]$
if $username \equiv \texttt{null}$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
let $PID_{RP} := parameters[PID_{RP}]$
let $Endpoint := parameters[Endpoint]$
if $PID_{RP} \notin \texttt{ListOfPID}() \vee Endpoint \notin \texttt{EndpointsOfRP}(PID)$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
let $UID := session[uid]$
let $PID_U := \texttt{ModPow}(PID_{RP}, UID, s'.p)$
let $Validity := \texttt{CurrentTime}() + s'.Validity$
let $Content := \langle PID_{RP}, PID_U, s'.ID, Validity \rangle$
let $Sig := \texttt{Sig}(Content, s'.SignKey)$
let $Token := \langle Content, Sig \rangle$
let $s'.Tokens := s'.Tokens +^{\langle\rangle} Token$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Token, Token \rangle \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
stop $\langle\rangle, s'$

E. 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 for $\langle ScriptUrl, p, q, PubKey \rangle$, where $ScriptUrl$ is the site to download IdP script, p and q are large prime defined before, and $PubKey$ is used to verify the IdP signed content.
- $sessions$ is same as it in RP process.
- $users$ is the set of RP registered user which is uniquely identified by the $Account$.

The new $functions$ are defined as follows

- $\texttt{ExEU}(a, q)$ is the Extended Euclidean algorithm, of which the result in RP process the $a^{-1} \mod q$.
- $\texttt{Random}()$ is a newly generated random number.
- $\texttt{RegisterUser}(Account)$ add the new user with $Account$ into RP's user list.

The relation of RP process R^r is shown as Algorithm 2.

Algorithm 2 R^r

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 $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript} \rangle$
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** $PID_{RP} := \texttt{ModPow}(s'.ID_{RP}, N_U, s'.IdP.p)$
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** $session := s'.sessions[cookie]$
24: **if** $session[state] \not\equiv expectRegistration$ **then**
25: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
26: **stop** $\langle f, a, m'\rangle, s'$
27: **end if**
28: **let** $RegistrationResult := body[RegistrationResult]$
29: **let** $Content := RegistrationResult.Content$
30: **if** $\texttt{checksig}(Content, RegistrationResult.Sig, s'.IdP.PubKey) \equiv \texttt{FALSE}$ **then**
31: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
32: **let** $session := \texttt{null}$
33: **stop** $\langle f, a, m'\rangle, s'$
34: **end if**
35: **if** $Content.Result \not\equiv OK$ **then**
36: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
37: **let** $session := \texttt{null}$
38: **stop** $\langle f, a, m'\rangle, s'$
39: **end if**
40: **let** $PID_{RP} := session[PID_{RP}]$
41: **let** $N_U := session[N_U]$
42: **let** $Nonce := \texttt{Hash}(N_U)$
43: **let** $Time := \texttt{CurrentTime}()$
44: **if** $PID_{RP} \not\equiv Content.PID_{RP} \vee Nonce \not\equiv Content.Nonce \vee Time > Content.Validity$ **then**
45: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
46: **let** $session := \texttt{null}$
47: **stop** $\langle f, a, m'\rangle, s'$
48: **end if**
49: **let** $session[PIDValidity] := Content.Validity$
50: **let** $Endpoint \in s'.Endpoints$
51: **let** $session[state] := expectToken$
52: **let** $Nonce' := \texttt{Random}()$
53: **let** $session[Nonce] := Nonce'$
54: **let** $Body := \langle PID_{RP}, Endpoint, Nonce'\rangle$
55: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, Body\rangle$
56: **stop** $\langle f, a, m'\rangle, s'$
57: **else if** $path \equiv /uploadToken$ **then**
58: **let** $cookie := headers[Cookie]$
59: **let** $session := s'.sessions[cookie]$
60: **if** $session[state] \not\equiv expectToken$ **then**
61: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
62: **stop** $\langle f, a, m'\rangle, s'$
63: **end if**
64: **let** $Token := body[Token]$
65: **if** $\texttt{checksig}(Token.Content, Token.Sig, s'.IdP.PubKey) \equiv \texttt{FALSE}$ **then**
66: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
67: **stop** $\langle f, a, m'\rangle, s'$

68: **end if**
69: **let** $PID_{RP} := session[PID_{RP}]$
70: **let** $Time :=$ `CurrentTime`()
71: **let** $PIDValidity := session[PIDValidity]$
72: **let** $Content := Token.Content$
73: **if** $PID_{RP} \not\equiv Content.PID_{RP} \vee Time > Content.Validity \vee Time > PIDValidity$ **then**
74: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle,$ `Fail`$\rangle$
75: **stop** $\langle f, a, m' \rangle, s'$
76: **end if**
77: **let** $PID_U := Content.PID_U$
78: **let** $t := session[t]$
79: **let** $Account :=$ `ModPow`$(PID_U, t, s'.IdP.p)$
80: **if** $Account \in$ `ListOfUser`() **then**
81: **let** `RegisterUser`$(Account)$
82: **end if**
83: **let** $session[user] := Account$
84: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle,$ `LoginSuccess`$\rangle$
85: **stop** $\langle f, a, m' \rangle, s'$
86: **end if**
87: **stop** $\langle\rangle, s'$

F. IdP scripting process

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

- $IdPDomain$ is the IdP's host.
- $Parameters$ is used to store the parameters received from other process.
- p is the large prime defined before.
- q is used to label the procedure point in the login.
- $refXHR$ is the nonce to mapping 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 if it exists and is visible 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 $pattern$.
- `RandonUrl`() returns a newly generated host string.

The relation of IdP scripting process $script\_idp$ is shown as Algorithm 3.

Algorithm 3 $script\_idp$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret \rangle$
1: **let** $s' := scriptstate$
2: **let** $command := \langle\rangle$
3: **let** $target :=$ `PARENTWINDOW`$(tree, docnonce)$
4: **let** $IdPDomain := s'.IdPDomain$
5: **switch** $s'.q$ **do**
6: **case** $start$**:**
7: **let** $N_U :=$ `Random`()
8: **let** $command := \langle$`POSTMESSAGE`$, target, \langle\langle N_U, N_U \rangle\rangle,$ `null`$\rangle$
9: **let** $s'.Parameters[N_U] := N_U$
10: **let** $s'.q := expectCert$
11: **case** $expectCert$**:**
12: **let** $pattern := \langle$`POSTMESSAGE`$, *, Content, *\rangle$
13: **let** $input :=$ `CHOOSEINPUT`$(scriptinputs, pattern)$
14: **if** $input \not\equiv$ `null` **then**
15: **let** $Cert := input.Content[Cert]$

```
16:        let s'.Parameters[Cert] := Cert
17:        if checksig(Cert.Content, Cert.Sig, s'.PubKey) ≡ null then
18:          let stop ⟨⟩
19:        end if
20:        let N_U := s'.Parameters[N_U]
21:        let PID_RP := ModPow(Cert.Content.ID_RP, N_U, s'.p)
22:        let s'.Parameters[PID_RP] := PID_RP
23:        let Endpoint := RandomUrl()
24:        let s'.Parameters[Endpoint] := Endpoint
25:        let Nonce := Hash(N_U)
26:        let Url := ⟨URL, S, IdPDomain, /dynamicRegistration, ⟨⟩⟩
27:        let s'.refXHR := Random()
28:        let command := ⟨XMLHTTPREQUEST, Url, POST,
               ⟨⟨PID_RP, PID_RP⟩, ⟨Nonce, Nonce⟩, ⟨Endpoint, Endpoint⟩⟩, s'.refXHR⟩
29:        let s'.q := expectRegistrationResult
30:      end if
31:    case expectRegistrationResult:
32:      let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
33:      let input := CHOOSEINPUT(scriptinputs, pattern)
34:      if input ≢ null then
35:        let RegistrationResult := input.Body[RegistrationResult]
36:        if RegistrationResult.Content.Result ≢ OK then
37:          let s'.q := stop
38:          let stop ⟨⟩
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:        let s'.Parameters[Endpoint_RP] := Endpoint_RP
52:        if Endpoint_RP ∉ Cert.Content.Endpoints ∨ PID_RP ≢ s'.Parameters[PID_RP] then
53:          let s'.q := stop
54:          let stop ⟨⟩
55:        end if
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** $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{GET}, \langle\rangle, s'.refXHR \rangle$
77: **let** $s'.q := expectToken$
78: **end if**
79: **end if**
80: **case** expectLoginResult**:**
81: **let** $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$
82: **let** $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
83: **if** $input \not\equiv \texttt{null}$ **then**
84: **if** $input.Body \not\equiv \texttt{LoginSuccess}$ **then**
85: **let stop** $\langle\rangle$
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** $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /authorize, \langle\langle PID_{RP}, PID_{RP}\rangle, \langle Endpoint, Endpoint\rangle, \langle Nonce, Nonce\rangle\rangle\rangle$
91: **let** $s'.refXHR := \texttt{Random}()$
92: **let** $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{GET}, \langle\rangle, s'.refXHR \rangle$
93: **let** $s'.q := expectToken$
94: **end if**
95: **case** expectToken**:**
96: **let** $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR \rangle$
97: **let** $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
98: **if** $input \not\equiv \texttt{null}$ **then**
99: **let** $Token := input.Body[Token]$
100: **let** $RPOringin := \langle s'.Parameters[Endpoint_{RP}], \texttt{S} \rangle$
101: **let** $command := \langle \texttt{POSTMESSAGE}, target, \langle Token, Token\rangle, RPOrigin \rangle$
102: **let** $s.q := stop$
103: **end if**
104: **end switch**
105: **let stop** $\langle s', cookies, localStorage, sessionStorage, command \rangle$

G. 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 same as them in IdP scripting process.

Here we define another new function $\texttt{SUBWINDOW}(tree, docnonce)$. This function takes the $tree$ define 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 not the opened windows, it will return the null.

The relation of RP scripting process $script\_rp$ is shown as Algorithm 4.

Algorithm 4 $script\_rp$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret \rangle$
1: **let** $s' := scriptstate$
2: **let** $command := \langle\rangle$
3: **let** $IdPWindow := \texttt{SUBWINDOW}(tree, docnonce).nonce$
4: **let** $RPDomain := s'.RPDomain$
5: **let** $IdPOringin := \langle s'.IdPDomian, \texttt{S} \rangle$
6: **switch** $s'.q$ **do**
7: **case** $start$**:**
8: **let** $Url := \langle \texttt{URL}, \texttt{S}, RPDomain, /login, \langle\rangle \rangle$
9: **let** $command := \langle \texttt{IFRAME}, Url, \_SELF \rangle$
10: **let** $s'.q := expectN_U$
11: **case** $expectN_U$**:**
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** $N_U := input.Content[N_U]$
16: **let** $Url := \langle \texttt{URL}, \texttt{S}, RPDomain, /startNegotiation, \langle\rangle \rangle$

```
        let s'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨N_U, N_U⟩⟩, s'.refXHR⟩
        let s'.q := expectCert
      end if
    case expectCert:
      let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        let Cert := input.Content[Cert]
        let command := ⟨POSTMESSAGE, IdPWindow, ⟨⟨Cert, Cert⟩⟩, IdPOringin⟩
        let s'.q := expectRegistrationResult
      end if
    case expectRegistrationResult:
      let pattern := ⟨POSTMESSAGE, *, Content, *⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        let RegistrationResult := input.Content[RegistrationResult]
        let Url := ⟨URL, S, RPDomain, /registrationResult, ⟨⟩⟩
        let s'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨RegistrationResult, RegistrationResult⟩⟩, s'.refXHR⟩
        let s'.q := expectTokenRequest
      end if
    case expectTokenRequest:
      let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        let PID_RP := input.Content.Body[PID_RP]
        let Endpoint := input.Content.Body[Endpoint]
        let Nonce := input.Content.Body[Nonce]
        let command := ⟨POSTMESSAGE, IdPWindow,
            ⟨⟨PID_RP, PID_RP⟩, ⟨Endpoint, Endpoint⟩, ⟨Nonce, Nonce⟩⟩, IdPOringin⟩
        let s'.q := expectToken
      end if
    case expectToken:
      let pattern := ⟨POSTMESSAGE, *, Content, *⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        let Token := input.Content[Token]
        let Url := ⟨URL, S, RPDomain, /uploadToken, ⟨⟩⟩
        let s'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨Token, Token⟩⟩, s'.refXHR⟩
        let s'.q := expectLoginResult
      end if
    case expectLoginResult:
      let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
      let input := CHOOSEINPUT(scriptinputs, pattern)
      if input ≠ null then
        if input.Body ≡ LoginSuccess then
          let LoadHomepage
        end if
      end if
  end switch
```