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

***Abstract*—Single sign-on (SSO) protocols such as OpenID Connect (OIDC), allow a user to maintain only the credential for an identity provider (IdP) to log into multiple relying parties (RPs). However, SSO introduces privacy threats, as (*a*) a curious IdP could trace a user's visits to RPs, and (*b*) colluding RPs could learn a user's online profile by linking her identities across these RPs. This paper presents a privacy-preserving SSO scheme, called UPPRESSO, to protect a user's online profile against both (*a*) an honest-but-curious IdP and (*b*) malicious RPs colluding with other users. An identity-transformation approach is proposed to generate untraceable ephemeral pseudo-identities for an RP and a user in each login, from which the visited RP derives a permanent account for the user, while these identity transformations provide unlinkability among the logins visiting different RPs. We built the UPPRESSO prototype providing OIDC-compatible SSO services, accessed from a commercial-off-the-shelf browser without installing any extension. The extensive evaluations show that it fulfills the security and privacy requirements of SSO with reasonable overheads.**

I. Introduction

Single sign-on (SSO) protocols such as OpenID Connect (OIDC) [**?**], OAuth 2.0 [**?**], and SAML [**?**], [**?**], are widely deployed for identity management and authentication. SSO allows a user to log into a website, known as the *relying party* (RP), using her account registered at another trusted web service, known as the *identity provider* (IdP). An RP delegates its user identification and authentication to the IdP, which issues an *identity token* (such as "id token" in OIDC and "identity assertion" in SAML) for a user to visit the RP. For instance, when an OIDC user initiates a login to visit an RP, the target RP constructs an identity-token request with its identity (denoted as ID_{RP}) and redirects it to a trusted IdP. After authenticating this user, the IdP issues an identity token binding the identities of the authenticated user and the target RP (i.e., ID_U and ID_{RP}), which is returned to the user and then forwarded to the RP. Finally, the RP verifies the identity token to determine whether the token holder is allowed to log in. Thus, a user maintains only one credential for user authentication to the IdP, instead of multiple credentials for visiting different RPs.

The wide adoption of SSO raises concerns on user privacy, because it facilitates the tracking of a user's login activities by interested parties [**?**], [**?**], [**?**], [**?**]. To issue identity tokens, an IdP should know the target RP to be visited by an authenticated user. So a curious IdP could trace a user's all login activities over time [**?**], [**?**], called *IdP-based login tracing* in this paper. Another privacy threat arises from the fact that RPs learn a user's identity from the identity tokens they receive. If an identical user identity is enclosed in the tokens for a user to visit different RPs, colluding RPs could link the logins across these RPs to learn the user's online profile [**?**], [**?**]. This threat is called *RP-based identity linkage*.

While preventing different privacy threats (i.e., IdP-based login tracing, RP-based identity linkage, or both), privacy-preserving SSO schemes [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] aim to provide services for users with a commercial-off-the-shelf (COTS) browser without installing any extensions. We analyze existing privacy-preserving SSO solutions in Section II, and also compare them with privacy-preserving identity federation [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].

This paper presents UPPRESSO, an Untraceable and Unlinkable Privacy-PREserving Single Sign-On protocol. We propose *identity transformations* in SSO, and integrates them into OIDC services. When visiting an RP in UPPRESSO, a user transforms ID_{RP} into ephemeral PID_{RP}, which is sent to a trusted IdP to transform ID_U into an ephemeral user pseudo-identity PID_U. Then, the identity token issued by the IdP binds only PID_U and PID_{RP}, instead of ID_U and ID_{RP}. On receiving the token, the RP transforms PID_U into an account unique at each RP but identical across multiple logins visiting this RP.

UPPRESSO prevents both IdP-based login tracing and RP-based identity linkage, while existing privacy-preserving SSO schemes address only one of them [**?**], [**?**], [**?**], [**?**], [**?**] or introduce another fully-trusted server in addition to the IdP [**?**], [**?**], [**?**]. That is, we offer an easy-to-deploy option to enhance the protections of user privacy for SSO services, for no extra server is introduced and the services are accessed from COTS browsers without extensions.

Our contributions are as follows.

- We proposed a novel identity-transformation approach for privacy-preserving SSO services and designed identity-transformation algorithms with desirable properties.
- We designed the UPPRESSO protocol by integrating the identity transformations into OIDC, and proved the security or privacy guarantees of SSO services.
- We implemented the UPPRESSO prototype on top of an open-source OIDC implementation. Through perfor-

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.[23|24]xxxx
www.ndss-symposium.org

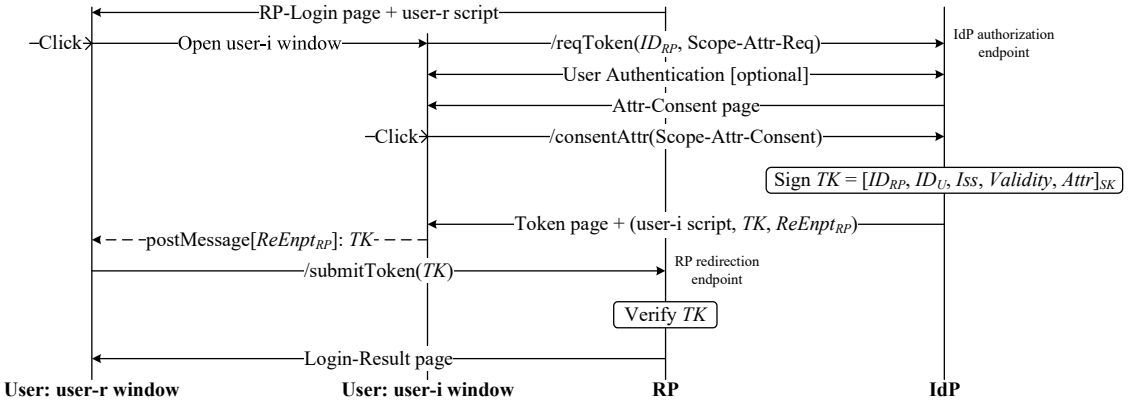


Fig. 1. The implicit SSO login flow of OIDC with pop-up UX

mance evaluations, we confirmed that UPPRESSO introduces reasonable overheads.

We explain the privacy threats in SSO and present related works in Section II. The identity-transformation approach is proposed in Section III, followed by the detailed designs of UPPRESSO in Section IV. Security and privacy are analyzed in Section V. We present the prototype implementation and evaluations in Section VI, and discuss extended issues in Section VII. Section VIII concludes this work.

II. Background and Related Work

A. OpenID Connect

OIDC [**?**] is one of the most popular SSO protocols. It supports different login flows: implicit flow, authorization code flow, and hybrid flow (a mix of the other two). The flows differ in the steps for requesting, receiving and forwarding identity tokens, but have common security requirements for the tokens. We present our designs in the implicit flow and discuss the support for the authorization code flow in Section VII.

Users and RPs register at an IdP with their identities and other information such as user credentials (e.g., passwords) and RP *redirection endpoints* [**?**] (i.e., the URLs to receive tokens, denoted as $ReEnpt_{RP}$). User operations in the login flows are conducted on a user agent (or browser typically).

To provide an in-context user experience (UX), the SSO login flow with pop-up UX [**?**], [**?**], [**?**] is recommended. As shown in Figure 1, when a user clicks the SSO login button in a browser window visiting the target RP (called the *user-r window* in this paper), an identity-token request is constructed with the RP's identity and the scope of requested user attributes. It pops up another window (i.e., the *user-i window*), and this request is sent to the IdP's *authorization endpoint* (i.e., the URL to issue tokens [**?**]). After authenticating the user, the IdP issues an identity token that encloses the (pseudo-)identities of the authenticated user and the target RP, the consented user attributes, a validity period, etc. The token is then forwarded to the RP's redirection endpoint via the user-i window. Finally, the RP verifies the token and allows the token holder to log in as the enclosed user (pseudo-)identity. There are scripts in two browser windows (called the *user-r script* and the *user-i script*), responsible for communicating with the respective origin web servers and also the cross-origin communications by the `postMessage` HTML5 API [**?**] within the browser.

Alternatively, if a browser works as the user agent of OIDC but with only one window, user attributes are consented in this window when redirected to the IdP [**?**], [**?**], [**?**], which is called *redirect UX*. After a token is issued by the IdP, it is sent to the RP's redirection endpoint as a fragment (but not parameters) of the URL due to security considerations [**?**]. So a script is downloaded from the RP to process the token.

B. Privacy-Preserving SSO and Identity Federation

In the original OIDC protocol an IdP signs identity tokens binding the identities of the authenticated user and the target RP [**?**], [**?**]. So a user's login activities are leaked due to these identities: The curious IdP learns the visited RP from ID_{RP} enclosed in an identity-token request (i.e., IdP-based login tracing), while colluding RPs link a user's logins visiting these RPs by ID_U in the tokens (i.e., RP-based identity linkage).

Privacy-preserving SSO. Existing solutions propose various mechanisms to hide (*a*) RP identities from the IdP or/and (*b*) user identities from the visited RPs. Table I compares these solutions, and only UPPRESSO provides SSO services accessed from COTS browsers while preventing both of the two privacy threats and not introducing extra trusted servers in addition to the honest IdP.

Pairwise pseudonymous identifiers (PPIDs) are specified [**?**], [**?**] and recommended [**?**] in OIDC to protect user privacy against colluding RPs. An IdP assigns a unique PPID for a user to log into every RP and encloses it in identity tokens, so colluding RPs cannot link the user by PPIDs. It does not prevent IdP-based login tracing because the IdP needs the RP's identity to assign PPIDs.

Other privacy-preserving schemes prevent IdP-based login tracing but leave users vulnerable to RP-based identity linkage, due to the unique user identities enclosed in identity tokens. For example, in BrowserID [**?**] after authenticating a user, an IdP issues a "user certificate" binding the user's identity to an ephemeral public key. The user then uses the corresponding

TABLE I
PRIVACY-PRESERVING SOLUTIONS OF SSO AND IDENTITY FEDERATION

	Solution	Privacy Threat‡		Extra Trusted Server†	Accounts Derived from User-maintained Secret‡
		IdP-based Login Tracing	RP-based Identity Linkage		
SSO	OIDC w/ PPID [?]	○	●	●	●
	BrowserID [?]	●	○	●	●
	SPRESSO [?]	●	○	◐[1]	●
	POIDC [?], [?]	●	○[2]	●	●
	MISO [?]	●	●	○	●
	UP-SSO [?]	●	●	●	○[3]
Identity Federation	PRIMA [?]	●	○	●	○
	PseudoID [?]	●	●	○	○
	Opaak [?]	●	●	●	○
	PP-IDF [?], [?], [?]	●	●	●	○
	Fabric Idemix [?]	●	●	●	○
SSO	UPPRESSO	●	●	●	●

‡ **Privacy Threat**: ● prevented, ○ not prevented. † **Extra Trusted Server**: ● no, ○ required.
‡ **Accounts Derived from User-maintained Secret**: ● no, ○ yes.

1. SPRESSO assumes a malicious IdP (but honest IdPs in others), and then a trusted forwarder server is needed to decrypt the RP identity and forward tokens to the RP.
2. A variation of POIDC [?] proposes to also hide a user's identity in the commitment and prove this to the IdP in zero-knowledge, but it takes seconds or more to generate such a zero-knowledge proof (ZKP) even for a powerful server [?], [?], [?], which is impracticable for a user agent in SSO.
3. UP-SSO [?] runs a user-side Intel SGX enclave with a secret to calculate PPIDs.

private key to sign a subsidiary "identity assertion" that binds the target RP's identity and sends both of them to the RP. In SPRESSO an RP creates a one-time tag (i.e., ephemeral pseudo-identity) for each login [?], and in POIDC [?], [?] a user sends an identity-token request with a hash commitment on the target RP's identity (but not ID_{RP}), which are enclosed in identity tokens along with the user's unique identity.

MISO [?] decouples the calculation of PPIDs from an honest IdP, to an extra fully-trusted mixer server that calculates a user's PPID based on ID_U, ID_{RP} and a secret after it receives the user's identity from the IdP. MISO prevents RP-based identity linkage for the RPs receives only PPIDs, and IdP-based login tracing for ID_{RP} is disclosed to the mixer but not the IdP. It protects user privacy against even collusive attacks by the IdP and RPs, but the mixer server could track a user's login activities. Similarly, UP-SSO [?] runs a fully-trusted Intel SGX enclave on the user side, which is remotely attested by the IdP and receives the secret to generate PPIDs.

Privacy-preserving identity federation. Identity federation enables a user registered at a trusted IdP to be accepted by other parties, potentially with different accounts, but browser extensions are installed to maintain a long-term user secret which is involved in the derivation of accounts. Although the term "single sign-on (SSO)" was used in some schemes [?], [?], [?], [?], [?], [?], this paper refers to them as *identity federation* to emphasize this difference.

In PRIMA [?], an IdP signs a credential that binds user attributes and a verification key. Using the corresponding signing key, the user authenticates herself and provides selected attributes to RPs. The verification key works as the user's identity and exposes her to RP-based identity linkage. PseudoID [?] introduces another trusted server in addition to the IdP, to blindly sign [?] a token that binds a long-term user secret and a pseudonym. The user then unblinds this token, presents it to the IdP which processes the secret, and then authenticates herself to an RP by this processed secret.

Privacy-preserving identity federation schemes are designed based on anonymous credentials [?], [?], [?]. In Opaak [?], UnlimitID [?], U-Prove [?], and EL PASSO [?], the IdP signs anonymous credentials, each of which binds a user secret, and a user proves ownership of the anonymous credentials using her secret. Similarly, Fabric [?] integrates Idemix anonymous credentials [?] for unlinkable pseudonyms.

Some privacy-preserving solutions of identity federation [?], [?], [?], [?], [?], [?] prevent both IdP-based login tracing and RP-based identity linkage, for (*a*) the RP's identity is not enclosed in the anonymous credentials and (*b*) the user selects different pseudonyms to visit different RPs. They even protect user privacy against collusive attacks by the IdP and RPs, as these pseudonyms cannot be linked through anonymous credentials [?], [?], [?] when the ownership of these credentials is proved to colluding RPs. In privacy-preserving identity federation this privacy protection depends on long-term user secrets to mask the relationship of among pseudonyms (or accounts), so a browser extension is always needed to maintain the user secrets. In Section VII we particularly discuss the collusion of the IdP and RPs in privacy-preserving SSO and identity federation.

Anonymous identity federation. Such approaches offer the strongest privacy, where a user visits RPs with pseudonyms that cannot be used to link any two actions. Anonymous identity federation was formalized [?] and implemented using cryptographic primitives such as group signature and ZKP [?], [?], [?]. Extended features including proxy re-verification [?], designated verification [?] and distributed IdP servers [?], are also considered.

These completely-anonymous authentication services only work for special applications and do not support user identification (or account uniqueness) at each RP, a common requirement in most applications.

C. OPRF-based Applications

PrivacyPass and TrustToken [?], [?] adopted the oblivious pseudo-random function (OPRF) protocol of Hash(EC)DH [?], [?] to generate anonymous tokens. A user generates a random number e to blind an unsigned token T into $[e]T$. After authenticating the user, a token server signs $[e]T$ using its private key k and returns $[ke]T$. The user then uses e to convert it into $(T, [k]T)$, which is redeemed when she anonymously accesses protected resources (see [?], [?] for details). In this procedure, the token server obliviously calculates a pseudo-random output as an anonymous token, because $(T, [k]T)$ and $([e]T, [ke]T)$ cannot be linked.

UPPRESSO employs a similar cryptographic technique.[1] A user selects t to transform ID_{RP} into $PID_{RP} = [t]ID_{RP}$. Then, the IdP uses the user's permanent identity u to calculate $PID_U = [u]PID_{RP} = [ut]ID_{RP}$, which is similar to token signing by the token server in PrivacyPass/TrustToken. Hence, IdP untraceability in UPPRESSO, i.e., the IdP cannot link ID_{RP} and $[t]ID_{RP}$, roughly corresponds to unlinkability of token signing-redemption in PrivacyPass/TrustToken, i.e., the token server cannot link T and $[e]T$.

UPPRESSO leverages this cryptographic technique in very different ways. First, it works among three parties, unlike the two-party PrivacyPass/TrustToken protocols. UPPRESSO shares the user-selected random number t with the target RP, enabling it to derive the user's permanent account, i.e., $Acct = [t^{-1}]PID_U$. This account is "obliviously" determined by the IdP when it calculates the user's pseudo-identity. Second, u and ID_{RP}, which roughly correspond to k and T in PrivacyPass/TrustToken, are assigned as the permanent identities of the user and the RP, respectively, to establish the relationship among identities and accounts. This is different from existing OPRF-based systems [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?] that always use k as a server's secret key. Finally, UPPRESSO leverages randomness of Hash(EC)DH OPRFs to ensure RP unlinkability, but this property is not utilized in PrivacyPass/TrustToken. This ensures colluding RPs cannot link any logins across RPs, even by sharing their knowledge about the pseudo-identities and permanent accounts in UPPRESSO. This property offers desirable indistinguishability of different users visiting colluding RPs. It roughly corresponds to the indistinguishability of different private keys for signing anonymous tokens, not considered in PrivacyPass/TrustToken.

Although UPPRESSO mathematically employs the same technique in the identity transformations as the Hash(EC)DH OPRF [?], [?], we require more properties of these algorithms. UPPRESSO essentially depends on the deterministicness property of pseudo-random functions to correctly derive the account for any t, obliviousness to ensure IdP untraceability, and pseudo-randomness to ensure RP unlinkability. In contrast, PrivacyPass and TrustToken [?], [?] depend on only the properties of deterministicness and obliviousness. Moreover, UPPRESSO requires additional properties for secure SSO services (i.e., user identification and RP designation; see Theorems 1 and 2 in Section V-B), which are not discussed in OPRFs [?], [?], [?]. Thus, an OPRF is not always ready to work as the identity transformations in UPPRESSO.

[1]We designed and implemented UPPRESSO in 2020, and in 2023 someone reminded us that the identity transformations are very similar to the Hash(EC)DH OPRF.

III. The Identity-Transformation Approach

In this section, we firstly list the security requirements of SSO, and present the identity-transformation approach.

A. Security Requirements for SSO Services

Non-anonymous SSO services are designed to allow a user to log into an honest RP as her unique account at this RP, by presenting identity tokens issued by a trusted IdP [?], [?], [?], [?], [?]. This security goal is achieved through the following properties of identity tokens.

First of all, *authenticity*, *confidentiality*, and *integrity* of identity tokens are necessary to prevent forging, eavesdropping and tampering. In SSO services, identity tokens are issued by the trusted IdP, transmitted over HTTPS/TLS [?], [?], [?], and finally forwarded to the target RPs by the authenticated user. Common security mechanisms within a browser (e.g., the mechanisms of HTTP session and `postMessage` targetOrigin) are also required for confidentiality of tokens [?], [?], [?], [?] (see Section IV-D for details).

Meanwhile, in each login the IdP issues an identity token that (*a*) specifies the target RP (i.e., *RP designation*) and (*b*) identifies the authenticated user who requests this token (i.e., *user identification*), as the enclosed (pseudo-)identities of RP and user [?], [?], [?] (see Section V-B for details). RP designation prevents malicious RPs from replaying received identity tokens to gain illegal access to other honest RPs as victim users. So an RP usually checks the RP (pseudo-)identity enclosed in a token before accepting it [?], [?], [?], [?], [?], [?], [?], [?], [?] and then allows the token holder to log in as the specified (pseudo-)identity (i.e., account).

B. Identity Transformations in SSO

As analyzed in Section II-B, existing privacy-preserving solutions of SSO hide only RP identities or user identities (i.e., only IdP-based login tracing or RP-based identity linkage is prevented), or introduce an extra fully-trusted server to process the (pseudo-)identities of both RP and user before an identity token is issued.

UPPRESSO attempts to *simultaneously* transform the identities of RP and user in each login, without extra trusted servers more than the honest-but-curious IdP, to provide untraceable and unlinkable SSO service. Table II lists the notations of the identity-transformation approach, and a sub/super-script (i.e., i, j, or l) may be omitted if it does not cause ambiguity.

First of all, each user owns her account unique at an RP. $\mathcal{F}_{Acct*}(ID_{U_i}, ID_{RP_j}) = Acct_{i,j}$ is proposed to determine a user's account at an RP. An account belonging to some user is *meaningful*, while a *meaningless* account at an RP does not belong to any user.

TABLE II
THE (PSEUDO-)IDENTITIES IN THE IDENTITY TRANSFORMATIONS

Notation	Definition	Lifecycle
ID_{U_i}	The i-th user's unique identity at the IdP.	Permanent
ID_{RP_j}	The j-th RP's unique identity at the IdP.	Permanent
$Acct_{i,j}$	The i-th user's account at the j-th RP.	Permanent
$PID^l_{U_i,j}$	The i-th user's pseudo-identity in her l-th login visiting the j-th RP.	Ephemeral
$PID^l_{RP_j}$	The j-th RP's pseudo-identity in the user's l-th login visiting this RP.	Ephemeral

To ensure IdP untraceability (or prevent IdP-based login tracing), a user initiates a login by calculating an *ephemeral* pseudo-identity $PID^l_{RP_j}$ for the target RP and sending an identity-token request for $PID^l_{RP_j}$ (but not ID_{RP_j}) to an IdP.

After authenticating the initiating user as ID_{U_i}, the IdP calculates an *ephemeral* $PID^l_{U_i,j}$ based on ID_{U_i} and $PID^l_{RP_j}$, and then issues an identity token that binds $PID^l_{U_i,j}$ and $PID^l_{RP_j}$. To ensure RP unlinkability (or prevent RP-based identity linkage), ID_{U_i} is not enclosed in the identity token.

On receiving a signed token, the RP calculates the user's *permanent* account (i.e., $Acct_{i,j}$) based on $PID^l_{U_i,j}$ and $PID^l_{RP_j}$, and allows the token holder to log in as $Acct_{i,j}$.

Therefore, in addition to $\mathcal{F}_{Acct*}(ID_U, ID_{RP}) = Acct$, we propose the following identity transformations in a login initiated by a user to visit an RP in privacy-preserving SSO:

- $\mathcal{F}_{PID_{RP}}(ID_{RP}) = PID_{RP}$, calculated by the initiating user, where ID_{RP} is the target RP's identity. In the IdP's view, $\mathcal{F}_{PID_{RP}}()$ is a one-way function and PID_{RP} is *indistinguishable* from random variables.
- $\mathcal{F}_{PID_U}(ID_U, PID_{RP}) = PID_U$, calculated by the IdP, where ID_U identifies the user requesting tokens. In the target RP's view, $\mathcal{F}_{PID_U}()$ is a one-way function and PID_U is *indistinguishable* from random variables.
- $\mathcal{F}_{Acct}(PID_U, PID_{RP}) = Acct$, calculated by the target RP. In the user's multiple logins visiting the RP, $Acct = \mathcal{F}_{Acct*}(ID_U, ID_{RP})$ is always derived.

The relationships among the (pseudo-)identities are depicted in Figure 2. Red and green blocks represent *permanent* identities and *ephemeral* pseudo-identities, respectively, and labeled arrows denote the transformations of (pseudo-)identities. PID_{RP} and PID_U are ephemeral; otherwise, the IdP could learn the visited RP. $PID_{U,j}$ for visiting different RPs are independent of each other; otherwise, colluding RPs could learn something on ID_U from $PID_{U,j}$s and link the logins.

In privacy-preserving SSO services integrating the identity transformations, an identity token *explicitly binds* $PID_{RP_j} = \mathcal{F}_{PID_{RP}}(ID_{RP_j})$ but *implicitly designates* only the RP with ID_{RP_j}. The following properties are required for the identity transformations:

- **Account Uniqueness.** Each user owns an account locally unique at an RP, determined by $\mathcal{F}_{Acct*}(ID_U, ID_{RP})$.
- **User Identification.** From an identity token, the designated honest RP derives only the account exactly identifying the user requesting this token.

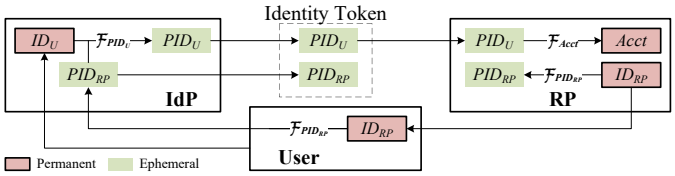


Fig. 2. Identity transformations in UPPRESSO

- **RP Designation.** An identity token is rejected, at any honest RPs other than the designated one.
- **IdP Untraceability.** The IdP learns nothing on the visited RP from the identity-token requests.
- **RP Unlinkability.** Colluding RPs cannot link any login initiated by an honest user visiting an RP, to any logins visiting any other colluding RPs by honest users.

In addition to the basic requirement of account uniqueness, user identification and RP designation ensure security of SSO services, while IdP untraceability and RP unlinkability protect user privacy (see Section V for the formal proofs).

IV. THE DESIGNS OF UPPRESSO

A. Threat Model

The system consists of an honest-but-curious IdP, as well as several honest or malicious RPs and users. This threat model is in line with widely-deployed SSO services in the Internet [**?**], [**?**], [**?**], [**?**].

Honest-but-curious IdP. The IdP strictly follows the protocols, while remaining interested in learning about a user's login activities. For example, it could store received messages to infer the relationship among ID_U, ID_{RP}, PID_U, and PID_{RP}. It never actively violates the protocols, so a script downloaded from the IdP also follows the protocols (see Section IV-D for specific designs for web applications).

The IdP maintains the private key well for signing identity tokens and RP certificates, and adversaries cannot forge such signed messages.

Malicious RPs. Adversaries could control a set of RPs by registering at the IdP as an RP or exploiting vulnerabilities to compromise some RPs. Malicious RPs could behave arbitrarily, attempting to compromise the security or privacy guarantees of UPPRESSO.

Malicious users. Adversaries could control a set of users by stealing their credentials or registering Sybil users in UPPRESSO. A malicious user could modify, insert, drop, or replay messages or even behave arbitrarily.

Colluding RPs and users. Malicious users and RPs could collude, attempting to break the security or privacy guarantees for honest users and RPs [**?**], [**?**], [**?**], [**?**]; that is, their objective is to (*a*) impersonate a victim user at honest RPs, (*b*) entice an honest user to log into an honest RP under another user's account, and (*c*) link an honest user's logins visiting colluding RPs. For example, they could manipulate PID_{RP} and t in a login, attempting to (*a*) entice honest users to return an identity token that could be accepted by some honest RP

or (*b*) affect the generation of PID_U to further analyze the relationship between ID_U and PID_U.

B. Assumptions

We assume secure communications between honest entities, and the cryptographic primitives are secure. The software stack of an honest entity (e.g., a browser) is correctly implemented to transmit messages to receivers as expected. These communication assumptions are realized by the commonly-used mechanisms in popular SSO services (e.g., HTTPS/TLS, HTTP session maintenance, and the `postMessage` HTML5 API in COTS browsers) [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].

While PID_{RP} but not ID_{RP} is processed by the IdP, it further requires that no other information about the visited RP is leaked to the IdP, in the requesting, receiving and forwarding of identity tokens. We realize this assumption by specific designs in Section IV-D. Our work focuses on the privacy threats introduced by SSO protocols, and does not consider the tracking of user activities by network traffic analysis or crafted web pages, as they can be prevented by other defenses.

UPPRESSO is designed for users who value privacy, or provides an effective option for such users. So privacy leakages due to re-identification by distinctive attributes across RPs are out of our scope. In UPPRESSO a user never consents to enclose distinctive attributes, such as telephone number, Email address, etc., in tokens or sets such attributes at any RP.

C. Identity-Transformation Algorithms

We propose the following identity transformations on an elliptic curve $\mathbb{E}$, where G is a generator on $\mathbb{E}$ of order n.

***ID_U, ID_{RP}* and *Acct*.** The IdP assigns a unique random integer $u \in \mathbb{Z}_n$ to a user (i.e., $ID_U = u$), and randomly selects unique $ID_{RP} = [r]G$ for an RP. Here, $[r]G$ denotes the addition of G on the curve r times. Then, $Acct_{i,j} = \mathcal{F}_{Acct*}(ID_{U_i}, ID_{RP_j}) = [ID_{U_i}]ID_{RP_j} = [u_i r_j]G$ is automatically assigned to a user at every RP.

A user's identity $ID_U = u$ is unknown to all entities except the honest IdP; otherwise, colluding RPs could calculate $[u]ID_{RP_j}$ for any known u and link these accounts. Meanwhile, $ID_{RP} = [r]G$ and $Acct = [ID_U]ID_{RP}$ are publicly-known, but r is always kept secret; otherwise, two colluding RPs with $ID_{RP_j} = [r]G$ and $ID_{RP_{j'}} = [r']G$ could link a user's accounts by checking whether $[r']Acct_j$ is equal to $[r]Acct_{j'}$ or not.

Fortunately, in UPPRESSO r is processed only once by the IdP, while u is used only by the IdP internally and not enclosed in any message; see Section IV-E for details.

***ID_{RP}-PID_{RP}* Transformation.** When visiting an RP, a user selects a random number $t \in \mathbb{Z}_n$ and calculates PID_{RP} as below. She also sends t to the RP.

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

***ID_U-PID_U* Transformation.** On receiving an identity-token request for PID_{RP} from a user authenticated as ID_U, the IdP calculates PID_U as below.

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

***PID_U-Acct* Transformation.** After verifying a signed identity token, the RP calculates $Acct$ as follows.

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

The above identity transformations ensure *account uniqueness*: At an RP, every user owns a unique account, because $ID_{RP} = [r]G$ is also a generator of $\mathbb{E}$ and then $Acct = \mathcal{F}_{Acct*}() = [ur]G$ is unique at this RP.

Meanwhile, in a login involving no adversary, the visited RP always derives the account belonging to the user who requests the identity token, by calculating $\mathcal{F}_{Acct}(PID_U, PID_{RP})$. From Equations 1, 2, and 3, it is derived that

$$Acct = [t^{-1}utr]G = [ur]G = \mathcal{F}_{Acct*}(ID_U, ID_{RP}) \quad (4)$$

In Section V, under the adversarial scenarios, we formally prove the other required properties (i.e., *user identification*, *RP designation*, *IdP untraceability*, and *RP unlinkability*) of the identity transformations.

D. The Designs Specific for Web Applications

When the required properties of identity transformations are satisfied, in UPPRESSO we need to additionally ensure:

- Authenticity, confidentiality, and integrity of identity tokens, by the common mechanisms in widely-deployed SSO services [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**],
- $PID_{RP} = [t]ID_{RP}$ is calculated based on the target RP's identity but not any other RPs' (i.e., the target RP is designated) and t is known only to the target RP and the initiating user (but not the honest IdP; otherwise, the IdP is able to infer ID_{RP} from PID_{RP} by itself),
- No direct privacy leakage in the requesting, receiving and forwarding of identity tokens (e.g., no HTTP `referer` header about the RP server is leaked to the IdP in HTTP requests).

In an original OIDC system with pop-up UX, as shown in Figure 1, a user requests a token by sending the target RP's identity (i.e., ID_{RP}) via the user-i window, and the IdP replies with the signed token and also the RP's URL to receive tokens (i.e., $ReEnpt_{RP}$). Then, after receiving the token, the user-i script forwards it to the user-r script by `postMessage` which is restricted by the origin of $ReEnpt_{RP}$ [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. The `postMessage` targetOrigin mechanism [**?**] restricts the recipient, so this ensures confidentiality of identity tokens because (*a*) only scripts downloaded from the origin of $ReEnpt_{RP}$ are legitimate recipients of the forwarded tokens and (*b*) the relationship between $ReEnpt_{RP}$ and ID_{RP} is maintained at the honest IdP server.

In UPPRESSO the IdP receives PID_{RP} (but not ID_{RP}) in an identity-token request, so it cannot reply with $ReEnpt_{RP}$ to the user-i window. An RP certificate, denoted as $Cert_{RP}$, is introduced to bind $ReEnpt_{RP}$ and ID_{RP}, signed by the IdP during the RP's initial registration.

Then, in each login, as shown in Figure 3, when the user-i window is popped up, the RP certificate (i.e., $Cert_{RP}$) and the scope of request user attributes are also carried with the

HTTP request to the IdP, which is generated by the user-r script. This HTTP request downloads the user-i script, which in turn verifies $Cert_{RP}$ to extract ID_{RP} and $ReEnpt_{RP}$, and calculates $PID_{RP} = [t]ID_{RP}$. Then, after the IdP issues a token, the user-i script forwards it to the user-r script by `postMessage`, which is restricted by the origin of $ReEnpt_{RP}$. This ensures that (*a*) $PID_{RP} = [t]ID_{RP}$ is calculated based on the visited RP's identity and (*b*) the token is forwarded to the corresponding RP server, because $ReEnpt_{RP}$ and ID_{RP} are signed in $Cert_{RP}$ by the IdP.

In the HTTP request to pop up the user-i window, $Cert_{RP}$ and the scope of requested user attributes are carried as *fragments* (but not parameters) [**?**], which are identified by `#` instead of `?`, so that they are not sent to the IdP web server but processed by the browser. Meanwhile, we save $ReEnpt_{RP}$ and t using the `sessionStorage` HTML object [**?**], because the user-i window is refreshed to finish user authentication and attribute consent. Therefore, $Cert_{RP}$, $ReEnpt_{RP}$ and t are processed locally within browsers.

On receiving an identity-token request, the IdP web server checks the included HTTP `origin` header to ensure it is sent by the user-i script, which verifies $Cert_{RP}$ to ensure the correct relationship between $ReEnpt_{RP}$ and ID_{RP}. Besides, the IdP's public key is set in the user-i script to verify $Cert_{RP}$, so a user configures nothing locally, like in other popular SSO systems [**?**], [**?**], [**?**], [**?**].

Compared with the original OIDC protocol in Figure 1, UPPRESSO adopts the common mechanisms for secure communications, except that the relationship between ID_{RP} and $ReEnpt_{RP}$ is maintained as $Cert_{RP}$ (i.e., in an offline way by the IdP). Thus, authenticity, confidentiality, and integrity of identity tokens are still ensured in UPPRESSO. Meanwhile, we implement the IdP's authorization endpoint in an OIDC-compatible way [**?**], [**?**]: When receiving a token request for an "unregistered" ID_{RP} (i.e., PID_{RP}), the IdP issues a token but with an "empty" $ReEnpt_{RP}$, and the calculation of PID_U is considered as a special way to assign PPIDs.

Finally, to prevent referer leakage in the HTTP request to pop up the user-i window, we need to ensure this request does not carry an HTTP `referer` header, which reveals the visited RP's domain to the IdP web server.[2] This is achieved by setting `<meta name="referrer" content="no-referrer">` in the RP's SSO login page [**?**]. In Figure 3 we can find that this HTTP request is the only communication from the user-r window to the IdP server.

E. The UPPRESSO protocol

System Initialization. $\mathbb{E}$, G and n are set up and publicly published. An IdP generates a key pair (SK, PK) to sign and verify identity tokens and RP certificates.

RP Registration. Each RP registers itself at the IdP to obtain ID_{RP} and its RP certificate $Cert_{RP}$ as follows. This may be conducted by face-to-face or online [**?**] means.

1. An RP pre-installs PK by trusted means. It submits a registration request, including the URL to receive identity tokens (i.e., $ReEnpt_{RP}$) and other information.
2. After examining the request, the IdP randomly selects $r \in \mathbb{Z}_n$ and assigns a *unique* point $[r]G$ to the RP as its identity. Note that r is not processed any more and then known to nobody due to the elliptic curve discrete logarithm problem (ECDLP). The IdP then signs $Cert_{RP} = [ID_{RP}, ReEnpt_{RP}, *]_{SK}$, where $[\cdot]_{SK}$ is a message signed using SK and $*$ is supplementary information such as the RP's unambiguous name.
3. The RP verifies $Cert_{RP}$ using PK, and accepts ID_{RP} and $Cert_{RP}$.

User Registration. Each user sets up her unique username and the corresponding credential for the IdP. The IdP assigns a unique random identity $ID_U = u \in \mathbb{Z}_n$ to the user.

It requires that ID_U is known *only* to the IdP. In UPPRESSO ID_U is used only by the IdP *internally*, not enclosed in any message. For example, a user's identity is generated and always restored by hashing her username concatenated with the IdP's private key. Then, ID_U is never stored in hard disks, and protected almost the same as the IdP's private key because it is *only* used to calculate PID_U as the IdP is signing an identity token binding $PID_U = [ID_U]PID_{RP}$.

Account Synchronization. Every RP regularly synchronizes all accounts at it from the IdP, and then locally maintains an up-to-date list of meaningful accounts. We discuss account synchronization in Section V-E.

SSO Login. A login flow involves three steps: Identity-token requesting, generation, and acceptance. In Figure 3, the IdP's and RP's operations are connected by two vertical lines, respectively. The user operations are split into two groups in different browser windows by vertical lines, one communicating with the IdP and the other with the RP. Solid horizontal lines indicate messages exchanged between a user and the IdP (or the RP), while dotted lines represent the mechanisms within a browser: (*a*) `postMessage` invocations between two scripts (or browser windows) or (*b*) `sessionStorage` objects accessed by refreshed pages of an HTTP session.

1. *Identity-Token Requesting.* The user requests an identity token for $PID_{RP} = [t]ID_{RP}$.

1.1 The target RP provides an SSO login page with the user-r script, and $Cert_{RP}$ is set in this script. The user clicks the SSO login button, to pop up another browser window responsible for communicating with the IdP.

1.2 The user-i window downloads the user-i script, which processes $Cert_{RP}$ and the scope of requested attributes sent from the user-r window. It verifies $Cert_{RP}$, extracts ID_{RP} and $ReEnpt_{RP}$ from $Cert_{RP}$, chooses a random number $t \in \mathbb{Z}_n$, and calculates $PID_{RP} = [t]ID_{RP}$.

1.3 The user-i script uses the `sessionStorage` HTML5 object to save $ReEnpt_{RP}$ and t, and then requests an identity token from the IdP by sending PID_{RP} and the scope of requested attributes.

2. *Identity-Token Generation.* The IdP calculates $PID_U = [ID_U]PID_{RP}$ and signs an identity token as follows.

[2] In original OIDC services this privacy leakage commonly exists, but does not matter because such services do not aim to prevent IdP-based login tracing.

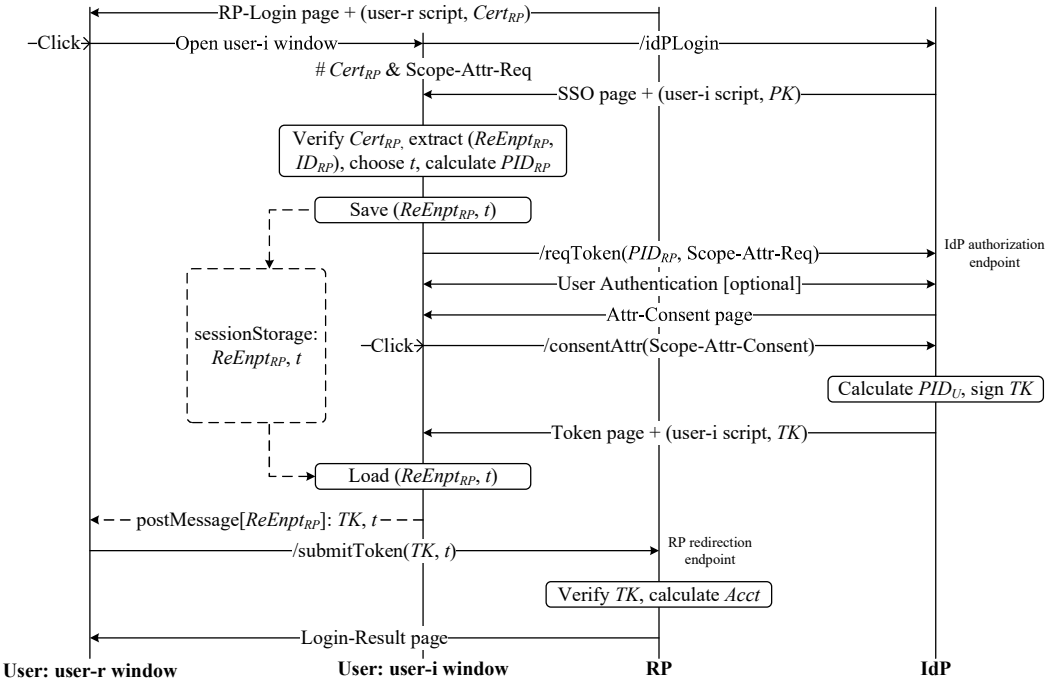


Fig. 3. The SSO login flow of UPPRESSO

2.1 On receiving an identity-token request, the IdP authenticates the initiating user as ID_U, if not authenticated yet. Then, it obtains the user's consent to enclose attributes.

2.2 The IdP calculates $PID_U = [ID_U]PID_{RP}$, and signs $TK = [PID_{RP}, PID_U, Iss, Validity, Attr]_{SK}$, where Iss identifies the IdP, $Validity$ indicates the validity period, and $Attr$ contains the user attributes.

2.3 The IdP replies with the identity token to the user-i window, as well as the user-i script to forward this token.

3. *Identity-Token Acceptance.* The RP receives the identity token and allows the user to log in.

3.1 From the `sessionStorage` object the user-i script loads $ReEnpt_{RP}$ and t, and forwards the identity token and t to the user-r window by `postMessage` which is restricted by the origin of $ReEnpt_{RP}$.

3.2 The user-r script submits it to the RP. The RP verifies the signature and the validity period of the received token, and calculates $Acct = [t^{-1}]PID_U$.

3.3 Only if $Acct$ is meaningful in its local account list, the RP allows the user to log in.

If any verification fails, the flow will be immediately terminated. For instance, when receiving invalid $Cert_{RP}$ the user halts it. The IdP rejects an identity-token request if the received PID_{RP} is not a point on $\mathbb{E}$, and the RP rejects a token if it is not well signed or a meaningless account is derived.

In Step 3.2, an RP does not check whether PID_{RP} enclosed in the signed token is equal to $[t]ID_{RP}$ or not, and this efficient design does not result in attacks, which is proved in Section V-B. It requires that in Step 3.3 the RP locally maintains an up-to-date account list, and account synchronization is explained with more details in Section V-E.

V. Security and Privacy Analysis

As (*a*) account uniqueness is analyzed in Section IV-C and (*b*) the designs in Section IV-D ensure secure communications and no direct privacy leakage in the communications, in this section we prove security (i.e., user identification and RP designation) and privacy (i.e., IdP untraceability and RP unlinkability) of the identity transformations in UPPRESSO.

A. Adversarial Scenarios

According to our design goals and the potential adversaries listed in Section IV-A, we consider three adversarial scenarios as below and the security and privacy guarantees are proved against different adversaries.

First of all, security (i.e., user identification and RP designation) is proved against *malicious RPs and users* in Section V-B. In the proofs, malicious entities could collude [**?**], [**?**], [**?**], attempting to (*a*) impersonate an honest user to log into some honest RP or (*b*) entice an honest user to log into an honest RP under another's account, by arbitrarily manipulating t and/or sending TK to an RP not designated.

Provided that the security properties are satisfied, we analyze the privacy properties *only* for successful logins because no meaningful account is derived in an unsuccessful login where the initiating user or the target RP deviates from the specifications. It makes no sense to track login activities resulting in meaningless accounts, and malicious RP and users cannot break privacy by deviating from the protocols. Thus, no malicious entities are considered in the proofs of privacy.

Therefore, (*a*) IdP untraceability is proved in Section V-C against *an honest-but-curious IdP*, which tries to infer the identities of the *honest* RPs being visited by an *honest* user,

and (*b*) RP unlinkability is analyzed in Section V-D against *a group of semi-honest RPs and users* which follow the protocols but share their information, attempting to link an honest user's accounts across these semi-honest RPs.

B. Security

In secure SSO systems, an identity token denoted as TK, which is requested by a user to visit an RP, *enables only this user to log into only the honest target RP as her account at this RP* (i.e., user identification and RP designation). It makes no sense to discuss the login results at malicious RPs.

Let's assume totally s users and p RPs in UPPRESSO, whose identities are denoted as $\mathbb{ID}_U = \mathbb{u} = \{u_{i;1\leq i\leq s}\}$ and $\mathbb{ID}_{RP} = \{[r_{j;1\leq j\leq p}]G\}$, respectively. There are meaningful accounts $Acct_{i,j} = [u_i]ID_{RP_j} = [u_i r_j]G$ at each RP. $Acct_{i,j}$ and ID_{RP_j} are publicly-known, while u_i and r_j are kept unknown to adversaries.

TK is signed by the IdP to bind PID_{RP} and PID_U, where (*a*) the RP with ID_{RP} is designated if $PID_{RP} = [t]ID_{RP}$ is calculated and (*b*) $PID_U = [ID_U]PID_{RP}$ is calculated based on the initiating user's identity ID_U.

User identification and RP designation are proved in Theorems 1 and 2, respectively, against malicious RPs colluding with users. Besides, as mentioned in Section IV-E, in Step 3.2 an (honest) RP directly derives accounts from any signed and unexpired identity token, without checking whether PID_{RP} in the token is equal to $[t]ID_{RP}$ or not. So in the proofs we do not assume any relationship among t, ID_{RP} and PID_{RP} which are received by an RP.

User identification means that, *from TK the designated honest RP derives only the meaningful account belonging to the user requesting TK*. We prove this property by showing that, at the RP designated by any TK, a meaningless account not belonging to the initiating user will never be derived.

THEOREM *1 (User Identification):* Given unknown $\mathbb{ID}_U$ and known $\mathbb{A}cct$, for any known $ID_{RP} \in \mathbb{ID}_{RP}$, malicious adversaries cannot find $\check{t}$ and TK binding $PID_{RP} = \mathcal{F}_{PID_{RP}}(ID_{RP}, \hat{t})$ and $PID_{\hat{U}} = \mathcal{F}_{PID_U}(ID_{\hat{U}}, PID_{RP})$ satisfying that $\mathcal{F}_{Acct}(PID_{\hat{U}}, \check{t}) = \mathcal{F}_{Acct*}(ID_{\check{U}}, ID_{RP})$, where $ID_{\hat{U}} \neq ID_{\check{U}}$ and $ID_{\hat{U}}, ID_{\check{U}} \in \mathbb{ID}_U$.

PROOF. It requires that, for any ID_{RP}, adversaries cannot find $\hat{t}$ and $\check{t}$ satisfying that $\mathcal{F}_{Acct}(PID_{\hat{U}}, \check{t}) = [\check{t}^{-1}]PID_{\hat{U}} = [\check{t}^{-1}ID_{\hat{U}}]PID_{RP} = [\check{t}^{-1}\hat{t}ID_{\hat{U}}]ID_{RP} = [ID_{\check{U}}]ID_{RP} = \mathcal{F}_{Acct*}(ID_{\check{U}}, ID_{RP})$, where $ID_{\hat{U}} \neq ID_{\check{U}}$.

When u_i and r_j are kept unknown, this is equivalent to the elliptic-curve discrete logarithm problem (ECDLP) of $\check{t}^{-1}\hat{t} = \log_{[\hat{u}r]G}[\check{u}r]G = \log_{\hat{A}cct}\check{A}cct$. □

RP designation means that, *from TK only the designated honest RP derives meaningful accounts*. We prove it by showing that, at any honest RPs other than the designated one, no meaningful account will be derived from TK.

THEOREM *2 (RP Designation):* When given known $\mathbb{ID}_{RP}$, known $\mathbb{A}cct$ and unknown $\mathbb{ID}_U$, malicious adversaries cannot find j_1, j_2, i_1, i_2, t_1, and t_2 which satisfy that $\mathcal{F}_{Acct}(\mathcal{F}_{PID_U}(ID_{U_{i_1}}, PID^*_{RP}), t_2) = \mathcal{F}_{Acct*}(ID_{U_{i_2}}, ID_{RP_{j_2}})$, where PID^*_{RP} may be calculated as $\mathcal{F}_{PID_{RP}}(ID_{RP_{j_1}}, t_1)$ or arbitrarily generated (i.e., RP_{j_1} or no RP is designated), $j_1 \neq j_2$, $1 \leq j_1, j_2 \leq p$, and $1 \leq i_1, i_2 \leq s$.

PROOF. If adversaries could find j_1, j_2, i_1, i_2, t_1 and t_2 which satisfy that $[t_2{}^{-1}t_1 ID_{U_{i_1}}]ID_{RP_{j_1}} = [ID_{U_{i_2}}]ID_{RP_{j_2}}$ where $j_1 \neq j_2$, the ECDLP of $t_2{}^{-1}t_1 = \log_{[u_{i_1}r_{j_1}]G}[u_{i_2}r_{j_2}]G = \log_{Acct_{i_1,j_1}} Acct_{i_2,j_2}$ would be solved.

The adversary might alternatively attempt to find PID^*_{RP}, j_2, i_1, i_2 and t_2 which satisfy that $[t_2{}^{-1}ID_{U_{i_1}}]PID^*_{RP} = [ID_{U_{i_2}}]ID_{RP_{j_2}}$, but this cannot be solved due to unknown ID_{U_i} when $i_1 \neq i_2$. Or, if $i_1 = i_2$, it would require the adversary to solve $[t_2{}^{-1}]PID^*_{RP} = ID_{RP_{j_2}}$ or $PID^*_{RP} = [t_2]ID_{RP_{j_2}}$. That is, the adversary chooses t_2, calculates $PID^*_{RP} = [t_2]ID_{RP_{j_2}}$, requests a token for PID^*_{RP}, and then logs into RP_{j_2} to "impersonate" himself. This is not an attack: RP_{j_2} is actually designated by TK and the adversary's account is derived at RP_{j_2}. □

C. Privacy against IdP-based Login Tracing

The honest IdP would attempt to infer the visited RPs, by collecting information from the identity-token requests. According to the designs and the detailed protocol in Sections IV-D and IV-E, it does not obtain any information about the target RP in a login (e.g., ID_{RP}, $ReEnpt_{RP}$ or $Cert_{RP}$), except the RP's ephemeral pseudo-identity PID_{RP}.

Next, we prove that, *PID_{RP} is indistinguishable from a random variable on $\mathbb{E}$ to the IdP*, and then it cannot (*a*) link multiple logins visiting an RP or (*b*) distinguish a login initiated by a user to visit any RP from those logins visiting other RPs by this user. Therefore, IdP untraceability is ensured.

THEOREM *3 (IdP Untraceability): The honest-but-curious IdP cannot distinguish $PID_{RP} = [t]ID_{RP}$ from a random variable on $\mathbb{E}$, where t is random in $\mathbb{Z}_n$ and kept unknown to the IdP.*

PROOF. As G is a generator on $\mathbb{E}$ of order n, $ID_{RP} = [r]G$ is also a generator of order n. As t is uniformly-random in $\mathbb{Z}_n$ and kept unknown, $PID_{RP} = [t]ID_{RP}$ is *indistinguishable* from a point that is uniformly-random on $\mathbb{E}$. □

This property actually has been proved in the Hash(EC)DH OPRF [**?**], [**?**] as obliviousness, where the OPRF server works similarly to the IdP and learns nothing about a client's inputs or outputs of the evaluated pseudo-random function (i.e., ID_{RP} or $Acct$ in UPPRESSO).

D. Privacy against RP-based Identity Linkage

Semi-honest RPs and users share their information, attempting to link an honest user's accounts across the semi-honest RPs. In each login, an RP obtains *only* a random number t and an identity token enclosing PID_{RP} and PID_U. From the token, it learns only an ephemeral pseudo-identity $PID_U = [ID_U]PID_{RP}$ of the initiating user, from which it derives a *permanent* locally-unique identifier (or account) $Acct = [ID_U]ID_{RP}$. It cannot directly calculate ID_U from PID_U or $Acct$ due to the ECDLP.

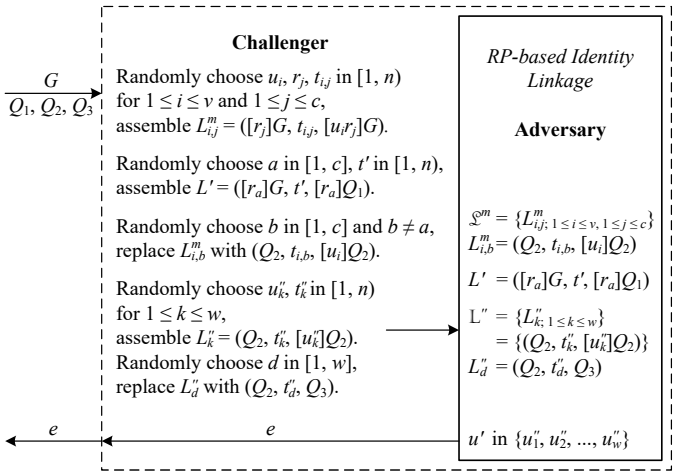


Fig. 4. The PPT algorithm $\mathcal{D}_R^*$ constructed based on the RP-based identity linkage game to solve the ECDDH problem.

With the trapdoor t, PID_{RP} and PID_U can be transformed into ID_{RP} and $Acct$, respectively, and vice versa. So we denote all the information that an RP learns in a login as a tuple $L = (ID_{RP}, t, Acct) = ([r]G, t, [ur]G)$.

When c RPs share information with each other, they create a shared view of all logins visiting any of them, denoted as $\mathbb{L}$. When they further collect information from v semi-honest users, the logins initiated by these users are picked out of $\mathbb{L}$ and linked together as $\mathfrak{L}^m = \begin{Bmatrix} L^m_{1,1}, & L^m_{1,2}, & \cdots, & L^m_{1,c} \\ L^m_{2,1}, & L^m_{2,2}, & \cdots, & L^m_{2,c} \\ \cdots, & \cdots, & L^m_{i,j}, & \cdots \\ L^m_{v,1}, & L^m_{v,2}, & \cdots, & L^m_{v,c} \end{Bmatrix}$, where $L^m_{i,j} = ([r_j]G, t_{i,j}, [u_i r_j]G) \in \mathbb{L}$ for $1 \le i \le v$ and $1 \le j \le c$. Any login in $\mathbb{L}$ but not linked in $\mathfrak{L}^m$ is initiated by an honest user to visit one of the c semi-honest RPs.

Next, in Theorem 4 we prove that, even when a number of semi-honest users share information with them, *the semi-honest RPs still cannot link any login visiting a semi-honest RP by an honest user to any other logins visiting other semi-honest RPs by honest users*. Then, RP unlinkability is ensured.

THEOREM *4 (RP Unlinkability): Given $\mathbb{L}$ and $\mathfrak{L}^m$, c semi-honest RPs and v semi-honest users cannot link any login visiting some of these RPs by an honest user to any subset of logins visiting other RPs by honest users.*

PROOF. Out of $\mathbb{L}$ we randomly choose a login $L' \neq L^m_{i,j}$ $(1 \le i \le v, 1 \le j \le c)$, which is initiated by an (unknown) honest user with $ID_{U'} = u'$ to a semi-honest RP_a where $a \in [1, c]$. Then, we randomly choose another semi-honest RP_b, where $b \in [1, c]$ and $b \neq a$. Consider any subset $\mathbb{L}'' \subset \mathbb{L}$ of w $(1 \le w < s - v)$ logins visiting RP_b by unknown honest users, and denote the identities of the users initiating these logins as $\mathbb{u}_w = \{u''_1, u''_2, \cdots, u''_w\} \subset \mathbb{u}$.

Next, we prove that the group of semi-honest adversaries cannot distinguish $u' \in \mathbb{u}_w$ from u' randomly selected in the universal set (i.e., $u' \in \mathbb{Z}_n$). So the adversaries cannot link L' to another login (or any subset of logins) visiting RP_b.

We define an RP-based identity linkage game $\mathcal{G}_R$ between an adversary and a challenger, to describe this login linkage: the adversary receives $\mathfrak{L}^m$, L', and $\mathbb{L}''$ from the challenger and outputs e, where (*a*) $e = 1$ if it decides u' is in $\mathbb{u}_w$ or (*b*) $e = 0$ if it believes u' is randomly chosen from $\mathbb{Z}_n$. Thus, the adversary succeeds in $\mathcal{G}_R$ with an advantage $\mathbf{Adv}$:

$$\begin{aligned} \mathrm{Pr}_1 &= \Pr(\mathcal{G}_R(\mathfrak{L}^m, L', \mathbb{L}'') = 1 \mid u' \in \mathbb{u}_w) \\ \mathrm{Pr}_2 &= \Pr(\mathcal{G}_R(\mathfrak{L}^m, L', \mathbb{L}'') = 1 \mid u' \in \mathbb{Z}_n) \\ \mathbf{Adv} &= |\mathrm{Pr}_1 - \mathrm{Pr}_2| \end{aligned}$$

As depicted in Figure 4, we design a PPT algorithm $\mathcal{D}_R^*$ based on $\mathcal{G}_R$ to solve the elliptic curve decisional Diffie-Hellman (ECDDH) problem: Given $(G, [x]G, [y]G, [z]G)$, decide whether z is equal to xy or randomly chosen in $\mathbb{Z}_n$, where G is a point on an elliptic curve $\mathbb{E}$ of order n, and x and y are integers randomly and independently chosen in $\mathbb{Z}_n$.

The algorithm $\mathcal{D}_R^*$ works as follows. (1) On receiving an input $(G, Q_1 = [x]G, Q_2 = [y]G, Q_3 = [z]G)$, the challenger chooses random numbers in $\mathbb{Z}_n$ to construct $\{u_i\}$, $\{r_j\}$, and $\{t_{i,j}\}$ for $1 \le i \le v$ and $1 \le j \le c$, with which it assembles $L^m_{i,j} = ([r_j]G, t_{i,j}, [u_i r_j]G)$. It ensures $[r_j]G \neq Q_2$ (or $r_j \neq y$) in this procedure. (2) It randomly chooses $a \in [1, c]$ and $t' \in \mathbb{Z}_n$, to assemble $L' = ([r_a]G, t', [r_a]Q_1) = ([r_a]G, t', [xr_a]G)$. (3) Next, the challenger randomly chooses $b \in [1, c]$ but $b \neq a$, and replaces ID_{RP_b} with $Q_2 = [y]G$. Hence, for $1 \le i \le v$, the challenger replaces $L^m_{i,b} = ([r_b]G, t_{i,b}, [u_i r_b]G)$ with $(Q_2, t_{i,b}, [u_i]Q_2) = ([y]G, t_{i,b}, [u_i y]G)$, and finally constructs $\mathfrak{L}^m$. (4) The challenger chooses random numbers in $\mathbb{Z}_n$ to construct $\{u''_k\}$ and $\{t''_k\}$ for $1 \le k \le w$, with which it assembles $\mathbb{L}'' = \{L''_{k;1\le k\le w}\} = \{(Q_2, t''_k, [u''_k]Q_2)\} = \{([y]G, t''_k, [u''_k y]G)\}$. It ensures $[u''_k]G \neq Q_1$ (i.e., $u''_k \neq x$) and $u''_k \neq u_i$, for $1 \le i \le v$ and $1 \le k \le w$. Finally, it randomly chooses $d \in [1, w]$ and replaces L''_d with $(Q_2, t''_d, Q_3) = ([y]G, t''_d, [z]G)$. Thus, $\mathbb{L}'' = \{L''_{k;1\le k\le w}\}$ represents the logins initiated by w honest users, i.e., $\mathbb{u}_w = \{u''_1, u''_2, \cdots, u''_{d-1}, z/y, u''_{d+1}, \cdots, u''_w\}$. (5) When the adversary of $\mathcal{G}_R$ receives $\mathfrak{L}^m$, L', and $\mathbb{L}''$ from the challenger, it returns e which is also the output of $\mathcal{D}_R^*$.

According to the above construction, x is embedded as $ID_{U'}$ of the login L' visiting the RP with $ID_{RP_a} = [r_a]G$, and z/y is embedded as $ID_{U''_d}$ of $\mathbb{L}''$ visiting the RP with $ID_{RP_b} = [y]G$, together with $\{u''_1, \cdots, u''_{d-1}, u''_{d+1}, \cdots, u''_w\}$. Meanwhile, $[r_a]G$ and $[y]G$ are two semi-honest RPs' identities in $\mathfrak{L}^m$. Because $x \neq u''_{k;1\le k\le w, k\neq d}$ and then x is not in $\{u''_1, \cdots, u''_{d-1}, u''_{d+1}, \cdots, u''_w\}$, the adversary outputs $s = 1$ and succeeds in this game only if $x = z/y$. Therefore, using $\mathcal{D}_R^*$ to solve the ECDDH problem, we have an advantage $\mathbf{Adv}^* = |\mathrm{Pr}_1^* - \mathrm{Pr}_2^*|$, where

$$\begin{aligned} \mathrm{Pr}_1^* &= \Pr(\mathcal{D}_R^*(G, [x]G, [y]G, [xy]G) = 1) \\ &= \Pr(\mathcal{G}_R(\mathfrak{L}^m, L', \mathbb{L}'') = 1 \mid u' \in \mathbb{u}_w) = \mathrm{Pr}_1 \\ \mathrm{Pr}_2^* &= \Pr(\mathcal{D}_R^*(G, [x]G, [y]G, [z]G) = 1) \\ &= \Pr(\mathcal{G}_R(\mathfrak{L}^m, L', \mathbb{L}'') = 1 \mid u' \in \mathbb{Z}_n) = \mathrm{Pr}_2 \\ \mathbf{Adv}^* &= |\mathrm{Pr}_1^* - \mathrm{Pr}_2^*| = |\mathrm{Pr}_1 - \mathrm{Pr}_2| = \mathbf{Adv} \end{aligned}$$

If the adversary of $\mathcal{G}_R$ has a non-negligible advantage, then $\mathbf{Adv}^* = \mathbf{Adv}$ is also non-negligible regardless of the security parameter λ. This violates the ECDDH assumption.

So the adversary has no advantage in $\mathcal{G}_R$ and cannot decide whether L' is initiated by some honest user with an identity in $\mathfrak{u}_w$ or not. Because RP_b is any semi-honest RP, this proof can be extended from RP_b to more semi-honest RPs. □

E. Account Synchronization

If new users are allowed to register after the system initialization, in order to distinguish meaningless accounts from meaningful ones in Step 3.3, an RP regularly synchronizes its accounts from the IdP. For example, when an account is derived but not in the visited RP's local account list, it contacts the IdP to synchronize accounts. After this instant account synchronization, the token holder will be rejected if the derived account is still considered as meaningless (i.e., not in the RP's local account list).

An RP cannot imperceptively treat the token holder with an account not in the list as a newly-registered user. Although a malicious user cannot log into this RP as any meaningful account belonging to others according to Theorems 1 and 2, she could log into such an RP as a meaningless but identical account in multiple logins. For example, by choosing ($\hat{t}_1$, $\check{t}_1$) and ($\hat{t}_2$, $\check{t}_2$) in two logins where $m = \hat{t}_1/\check{t}_1 = \hat{t}_2/\check{t}_2$, as analyzed in the proof of Theorem 1. this malicious user would actually own another account $Acct = [mID_U]ID_{RP} \neq [ID_U]ID_{RP}$.

The instant account synchronization can be replaced by conditional PID_{RP} checking as below. If an RP derives some account not in its local account list, it additionally checks PID_{RP} in this case: If PID_{RP} enclosed in the signed token is equal to $[t]ID_{RP}$, this account is asserted to belong to a newly-registered user and then the RP updates its account list locally; otherwise, it rejects this token.

VI. Implementation and Evaluation

We implemented the UPPRESSO prototype[3] and conducted experimental comparisons with two open-source SSO systems: (*a*) MITREid Connect [**?**], a PPID-enhanced OIDC system with redirect UX, preventing only RP-based identity linkage, and (*b*) SPRESSO [**?**], which prevents only IdP-based login tracing. All these solutions work with COTS browsers as user agents, while UPPRESSO and SPRESSO implement OIDC-compatible services with pop-up UX.

A. Prototype Implementation

The UPPRESSO prototype implemented the identity transformations on the NIST P256 elliptic curve where $n \approx 2^{256}$, and the IdP was developed on top of MITREid Connect [**?**], with minimal code modifications. All systems employ RSA-2048 and SHA-256 to generate tokens. The scripts of user-i and user-r consist of about 180 and 80 lines of JavaScript code, respectively. The cryptographic computations such as $Cert_{RP}$ verification and PID_{RP} negotiation are conducted based on elliptic [**?**], an open-source JavaScript library.

We developed a Java-based RP SDK with about 310 lines of code on the Spring Boot framework. The main function encapsulates the RP operations of UPPRESSO: verify an identity token and derive $Acct$. The cryptographic computations are finished using the Spring Security library. Then, an RP can invoke necessary functions by adding less than 10 lines of Java code, to access the services provided by UPPRESSO.

SPRESSO implements all entities by JavaScript based on node.js, while MITREid Connect provides Java implementations of IdP and RP SDK. Thus, in UPPRESSO and MITREid Connect, we implemented RPs based on Spring Boot by integrating the respective SDKs. In all three schemes, the RPs provide the same function of obtaining the user's account from a verified identity token.

B. Performance Evaluation

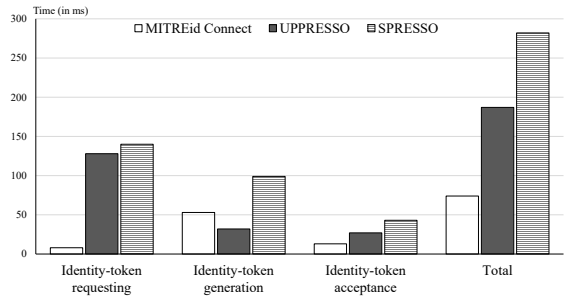
Experiment setting. We conducted experiments in two settings: (*a*) on a PC with AMD Ryzen 7 57000X CPU and 32GB memory, a browser, the IdP, the visited RP, and the extra forwarder server of SPRESSO were deployed on separated virtual machines, each of which ran Windows Server 2025 with 2 vCPUs and 4GB RAM, and (*b*) the browser running locally on the PC with AMD Ryzen 7 57000X CPU and 32GB memory, remotely accessed the servers (IdP and RP servers, and the forwarder server of SPRESSO) deployed on the Alibaba Cloud Elastic Compute Service, each of which ran Windows Server 2025 with 8 vCPUs and 32GB RAM. The first setting ran all functions on separated virtual machines but within the same physical PC, to minimize the impact of network delays.

Comparisons. We split the login flow into three phases for detailed comparisons: (1) *identity-token requesting* (Step 1 in Figure 3), to construct an identity-token request and send it to the IdP server; (2) *identity-token generation* (Step 2), to generate an identity token at the IdP server, while the user authentication and the user-attribute consent are excluded; and (3) *identity-token acceptance* (Step 3), where the RP receives, verifies, and parses the identity token.

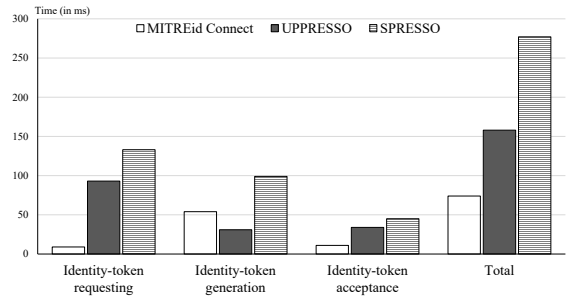
The IdP and the RP of all solutions, as well as the forwarder server in SPRESSO, require user-side scripts, which introduces additional time for downloading. In the identity-token requesting phase of UPPRESSO a browser downloads the user-i script, as described in Section IV-D. As mentioned in Section II-A, to process the token retrieved from the IdP, in MITREid Connect a user-r script is downloaded during the phase of token generation. In addition to a script from the IdP in identity-token requesting, SPRESSO needs another script from the forwarder server in the token acceptance phase [**?**]. Fortunately, once a browser visits a web server, it caches the scripts locally, reducing the download time for subsequent visits. This caching is enabled by the `localStorage` HTML object [**?**] and HTTP `Cache-Control` header.

We measured the time cost for both initial and subsequent visits (1,000 trials) and compared the average SSO login times

[3]The prototype is open-sourced at https://github.com/uppresso/.

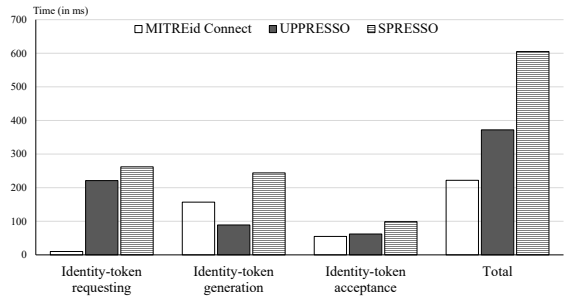


[i] Initial visit

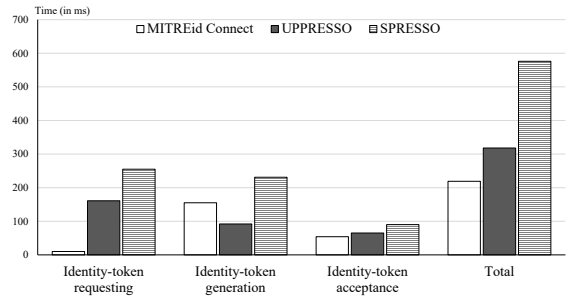


[ii] Subsequent visit

(*a*) Within a local virtual environment



[i] Initial visit



[ii] Subsequent visit

(*b*) With a remotely-visiting browser

Fig. 5. The time costs of SSO login in MITREid Connect, UPPRESSO, and SPRESSO

of three schemes. As shown in Figure 5, MITREid Connect, UPPRESSO, and SPRESSO require (*a*) 74 ms, 187 ms, and 282 ms for initial visits, and 69 ms, 158 ms and 277 ms for subsequent visits, or (*b*) 222 ms, 372 ms, and 605 ms for initial visits, and 219 ms, 318 ms and 576 ms for subsequent visits.

Regarding identity-token requesting, the RP of MITREid Connect immediately constructs an identity-token request. UPPRESSO incurs overheads in opening a new browser window and downloading the user-i script. In SPRESSO the user also needs to open a new browser window, and obtain information about the IdP and encrypt the visited RP's domain using an ephemeral key, resulting in extra overheads.

UPPRESSO requires the least time for generating identity tokens for it receives the token from the IdP without additional processing. In this phase MITREid Connect and SPRESSO require extra time as the browser downloads a script from the RP and the forwarder, respectively. After receiving an identity token from the IdP, in SPRESSO the user decrypts the RP's domain by scripts and sends it to the RP's redirection endpoint. Moreover, SPRESSO takes slightly more time to generate an identity token, as it implements the IdP using node.js and uses a JavaScript cryptographic library that is a little less efficient than the Java library used in the others.

Finally, in the identity-token acceptance phase, three solutions take similar amounts of time for the RP to receive a token and accept it.

By comparing the average time costs of initial visits and subsequent visits, we find that the performances of all schemes are improved and the improvement is the most significant in UPPRESSO for the size of its scripts is the largest (134kB vs. 2.34kB in MITREid Connect and 6.84kB in SPRESSO).

VII. Discussions

IdP-RP collusive attacks. This work prefers to provide better user experience rather than prevent IdP-RP collusive attacks. Even when the IdP strictly follows the protocols but shares information with RPs, a user would complete her logins *entirely* with colluding entities, and then the IdP and RPs could always link a user's (pseudo-)identities and accounts (i.e., both IdP untraceability and RP unlinkability are broken), unless a secret, unknown to the colluding IdP and RPs, is introduced to mask the relationship among the (pseudo-)identities and accounts. That is, regardless of which $\mathcal{F}_{Acct*}()$ is adopted to determine $Acct_{i,j} = \mathcal{F}_{Acct*}(ID_{U_i}, ID_{RP_j})$, the IdP could always collude with RPs to track a user's login activities by analyzing the relationship between the user's identity and accounts derived at the RPs, unless $Acct_{i,j} = \mathcal{F}_{Acct*}(s_i, ID_{U_i}, ID_{RP_j})$ is adopted and s_i is a long-term secret known to only the user. In this case, a user needs to install a browser extension to maintain the secret, and privacy-preserving identity federation [**?**], [**?**], [**?**], [**?**], [**?**] prefers stronger privacy than user experience. If this secret is lost or leaked, the user has to notify all RPs to update her accounts derived from this secret [**?**], [**?**].[4] Moreover, if

[4]This user-maintained secret used to derive accounts (or mask the relationship of identities and accounts) is different from the credential for user authentication to the IdP: If the credential is lost or leaked, the user only needs to update it at the IdP and her accounts are kept unchanged.

the user accesses the services from different devices, addition operations are required to synchronize the secret among these devices.

Alternatively, MISO [**?**] introduces an extra fully-trusted mixer server other than the IdP to keep s_i, to calculate user pseudo-identities (or accounts) in each login. Then, in MISO the mixer server can track all users' login activities, but the IdP cannot even when colluding with RPs.

On the contrary, UPPRESSO is designed to provide privacy-preserving SSO services accessed from COTS browsers without extra trusted servers. Our strategy offers a different option. In the future, we plan to improve the identity-transformation algorithms to process such a user secret as optional arguments, and then flexible privacy-preserving services will be provided.

Support for the authorization code flow. In the authorization code flow of OIDC [**?**], the IdP does not directly return identity tokens. Instead, it generates an authorization code, which is forwarded to the target RP. The RP uses this code to retrieve identity tokens from the IdP.

$\mathcal{F}_{Acct*}()$, $\mathcal{F}_{PID_U}()$, $\mathcal{F}_{PID_{RP}}()$ and $\mathcal{F}_{Acct}()$, can be also integrated into the authorization code flow of OIDC to transform (pseudo-)identities in signed identity tokens. Then, in the flow an authorization code (but not the token) will be sent to the user-r script, and to the RP finally. This code serves as an index to retrieve tokens by the RP, not disclosing any information about the user. On receiving an authorization code, the RP uses it as well as a credential issued by the IdP during the initial registration [**?**], to retrieve identity tokens from the IdP's token endpoint [**?**].

Meanwhile, to hide RP identities from the IdP in the retrieval of identity tokens, privacy-preserving credentials (e.g., ring signatures [**?**] or privacy passes [**?**], [**?**]) and anonymous communications (e.g., oblivious proxies [**?**] or Tor [**?**]) need to be adopted for RPs. Otherwise, if the visited RP is not anonymously authenticated to the IdP in the retrieval of tokens, IdP untraceability is broken.

Alternatives to generate ID_{RP} and bind $ReEnpt_{RP}$. In UPPRESSO the IdP generates random ID_{RP} and signs an RP certificate to bind ID_{RP} and $ReEnpt_{RP}$, which is verified by the user-i script. This ensures the relationship between the RP designated by an identity token and the URL to receive this token. It also guarantees that the target RP has already registered itself at the IdP and prevents unregistered RPs from utilizing the IdP's services [**?**], [**?**].

An alternative method for binding ID_{RP} and $ReEnpt_{RP}$ is to design a *deterministic* scheme to calculate unique ID_{RP} based on the RP's unambiguous name such as its domain. This can be achieved by encoding the domain with a hashing-to-elliptic-curves function [**?**], which provides collision resistance but not revealing the elliptic-curve discrete logarithm of the output. It generates a point on $\mathbb{E}$ as ID_{RP}, ensuring the *uniqueness* of $ID_{RP} = [r]G$ while keeping r unknown. In this case, in Step 1.2 the user-r script sends only the RP's redirection endpoint (i.e., its URL for receiving tokens) but not its RP certificate, and the user-i script calculates ID_{RP} by itself based on the RP's domain. Then, any RP can be visited through the SSO services, either registered or not.

However, if some RP changes its domain, for example, from `https://theRP.com` to `https://RP.com`, the accounts (i.e., $Acct = [ID_U]ID_{RP}$) will inevitably change. Thus, a user is required to perform special operations to migrate her account to the updated RP system. It is worth noting that the user operations cannot be eliminated in the migration to the updated one; otherwise, it implies two colluding RPs could link a user's accounts across these RPs.

Restriction of the user-r script's origin. The user-i script forwards tokens to the user-r script, and the `postMessage` targetOrigin mechanism [**?**] restricts the recipient of the forwarded identity tokens, to ensure that the tokens will be sent to the intended $ReEnpt_{RP}$, as specified in the RP certificate. The targetOrigin is specified as a combination of protocol, port (if not present, 80 for `http` and 443 for `https`), and domain (e.g., `RP.com`). The user-r script's origin must accurately match the targetOrigin to receive tokens.

The targetOrigin mechanism does not check the whole URL path in $ReEnpt_{RP}$, but introduces no *additional* risk. Consider two RPs in one domain but receiving tokens through different URLs, e.g., `https://RP.com/honest/tk` and `https://RP.com/malicious/tk`. This mechanism cannot distinguish them. Because a browser controls access to web resources following the same-origin policy (SOP) [**?**], a user's resources in the honest RP server is accessible to the malicious server. For example, it could steal cookies by `window.open('https://RP.com/honest').document.cookie`, even if the honest RP restricts only the HTTP requests to specific paths are allowed to access its cookies. So this risk is caused by the SOP model of browsers but not our designs, and commonly exists in SSO services accessed from COTS browsers [**?**], [**?**], [**?**], [**?**], [**?**].

Support for the redirect UX login. UPPRESSO can be adapted to support the OIDC login flow with Redirect UX (full-page navigation) instead of the pop-up UX. In the standard OIDC Redirect UX, the Relying Party (RP) and Identity Provider (IdP) typically exchange parameters via HTTP 302 redirections. However, to maintain UPPRESSO's privacy guarantees (i.e., IdP-based login tracing), the flow is modified to ensure the RP's identity remains hidden from the curious IdP during these navigational hand-offs.

1. Initiating the Login (RP to IdP): When the user initiates a login, the RP redirects the browser to the IdP. Unlike standard OIDC, which passes the client_id and redirect_uri as query parameters, UPPRESSO must pass the RP Certificate ($Cert_{RP}$) and scope as URL fragments (following the # symbol) . This ensures these sensitive parameters are processed only by the browser and not sent to the IdP's web server. Crucially, the RP must also suppress the HTTP Referer header (e.g., using rel="noreferrer") during this redirection. This prevents the IdP server from reading the RP's domain from the HTTP request headers.

2. Authenticating and Returning (IdP to RP): Same as in pop-up UX, the UPPRESSO client-side script (formerly

the user-i script) verifies $Cert_{RP}$, calculates the ephemeral pseudo-identity PID_{RP} locally , and requests the token from the IdP via AJAX/Fetch. In standard OIDC, the IdP server would issue an HTTP 302 redirect to return the user to the RP. However, in UPPRESSO, the IdP server does not know the RP's redirection endpoint ($ReEnpt_{RP}$) . Therefore, the IdP returns the signed token to the user-i script within the current page. This script then retrieves the valid $ReEnpt_{RP}$ and t from the local session and performs a client-side redirection (e.g., window.location.href) to navigate the user back to the RP with the token in the URL fragment. This approach ensures that neither the entry redirection nor the return redirection exposes the RP's permanent identity or location to the curious IdP.

VIII. Conclusion

This paper presents UPPRESSO, an untraceable and unlinkable privacy-preserving SSO system for protecting a user's online profile across RPs against both a curious IdP and colluding RPs. We propose an identity-transformation approach and design algorithms satisfying the requirements of security and privacy: (*a*) $\mathcal{F}_{Acct*}()$ determines a user's account, unique at an RP and unlinkable across RPs, (*b*) $\mathcal{F}_{PID_{RP}}()$ protects a visited RP's identity from the curious IdP, (*c*) $\mathcal{F}_{PID_U}()$ prevents colluding RPs from linking a user's multiple logins visiting different RPs, and (*d*) $\mathcal{F}_{Acct}()$ derives a user's permanent account from ephemeral pseudo-identities. The identity transformations are integrated into the widely-adopted OIDC protocol, maintaining user convenience and security guarantees of SSO services. Our experimental evaluations of the UPPRESSO prototype demonstrate its efficiency. For example, when the scripts are not cached, in UPPRESSO a login takes 187 ms in average when the IdP, the visited RP, and user browsers are deployed in a virtual private cloud, or 372 ms when a user visits remotely. Compared with MITREid Connect, it introduces an overhead of 89 – 150 ms to prevent two privacy threats.