UPRESSO: An Unlinkable Privacy-REspecting Single Sign-On System

***Abstract*—The Single Sign-On (SSO) service, provided by identity provider (IdP), is widely deployed and integrated to bring the convenience to both the relying party (RP) and the users. However, the privacy leakage is an obstacle to the users' adoption of SSO, as the curious IdP may track at which RPs users log in, while collusive RPs could link the user from a common or related identifer(s) issued by the IdP. Existing solutions preserve the user's privacy from either the curious IdP or the collusive RPs, but never from the both entities. In this paper, we provide an SSO system, named UPRESSO, to hide the user's accessed RPs from the curious IdP and prevent the identity linkage from the collusive RPs. In UPRESSO, IdP generates a different privacy-preserving ID (PID_U) for a user among RPs, and binds PID_U with a transformation of the RP identifier (PID_{RP}), without obtaining the real RP identifier. Each RP uses a trapdoor to derive the user's unique account from PID_U, while a user's accounts are different among the RPs. UPRESSO is compatible with OpenID Connect, a widely deployed and well analyzed SSO system; where dynamic registration is utilized to make PID_{RP} valid in the IdP. The analysis shows that user's privacy is preserved in UPRESSO without any degradation on the security of OpenID Connect. We have implemented a prototype of UPRESSO. The evaluation demonstrates that UPRESSO is efficient, and only needs 208 ms for a user to login at an RP in our environment.**

***Index Terms*—Single Sign-On, security, privacy, trace, linkage**

I. Introduction

Single sign-on (SSO) systems, such as OAuth [**?**], OpenID Connect [**?**] and SAML [**?**], have been widely adopted nowadays as a convenient web authentication mechanism. SSO delegates user authentication from websites, so-called relying parties (RPs), to a third party, so-called identity providers (IdPs), so that users can access different services at cooperating sites via a single authentication attempt. Using SSO, a user no longer needs to maintain multiple credentials for different RPs, instead, she maintains only the credential for the IdP, who in turn will generate corresponding *identity proofs* for those RPs. Moreover, SSO shifts the burden of user authentication from RPs to IdPs and reduces security risks and costs at RPs. As a result, SSO has been widely integrated with modern web systems. We analyze the Alexa top-100 websites [**?**] and find 80% of them support SSO service, and the analysis in [**?**] identifies SSO support on 6.30% of the Alexa top 1 million websites. Meanwhile, many email and social networking providers (such as Google, Facebook, Twitter, etc.) have been actively serving as social identity providers to support social login.

SSO systems need to provide secure authentication [**?**], that is to ensure an honest user logs in to an honest RP under the correct identity (i.e., account). To achieve this, the identity proof should be valid only for the RP that the user requests to log in to (i.e., binding), and never leaked to other entities except this RP and the user (i.e., confidentiality); while this RP should never accept any information from the corrupted identity proof (i.e., integrity). However, various attacks are found to exploit the vulnerabilities in the SSO systems to break at least one of these three principles [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], and the adversary could impersonate the victim user at an RP or log in the browser of an honest user under an adversary's identity (i.e., identity injection). For example, Friendcaster was found to blindly accept any received identity proof [**?**], [**?**] (i.e., not checking binding), then a malicious RP could obtain the identity proof from a user who attempts to log in to this malicious RP, and use it to log in to Friendcaster as the victim user [**?**]; some RPs of Google ID SSO were found to accept the user's attributes unprotected in the identity proof (i.e., out scope of integrity protection), and therefore a malicious user could add incorrect attributes (e.g., the email address) in identity proof and then act as any user at the RP [**?**].

The wide adoption of SSO also raises new privacy concerns regarding online user tracking and profiling [**?**], [**?**]. Privacy leakage exists in all current SSO protocols and implementations. We adopt the SSO authentication session in the OpenID Connect (OIDC) as an example, to figure out the leakage. As shown in Fig. **??**, on receiving a login request from a user (Step 1), the RP uses the RP's identifier to construct an authentication request and redirects it to IdP (Step 2); then, the IdP completes the user's authentication in Step 3, generates an identify proof for the user and binds it with the RP in Step 4, and redirects the identity proof to the RP confidentially in Step 5; finally, the RP verifies the binding and integrity of identity proof and sends the result to the user. From the authentication session, we find collusive RPs and curious IdP could break the user's privacy as follows.

- *RP-based identity linkage*. If the common (or derivable from others) identifiers are used in the identity proofs for a same user across different RPs, which is the case even in several widely deployed SSO systems [**?**], [**?**], collusive RPs could not only track her online traces but also correlate her attributes across the sites [**?**].
- *IdP-based access tracing*. IdP obtains the user's unique identifer in Step 3 and the identifers of the visited RPs in Step 2. Therefore, a curious IdP could easily discover all RPs accessed by a user and reconstruct her access traces.

Meanwhile, large IdPs, especially social IdPs like Google and Facebook, are known to be interested in collecting

users' online behavioral information for various purposes (e.g., Screenwise Meter [**?**], Onavo [**?**]). By simply serving the IdP role, these companies can easily collect a large amount of continuous data to reconstruct users' online traces. Moreover, many service providers are also hosting a variety of web services, which makes them easy to link the same user's multiple logins in each RP as the user's unique identifier is contained in the identity proof. Through internal integration, they could obtain rich information from SSO data to profile their clients.

While the privacy problems in SSO have been widely recognized [**?**], [**?**], only a few solutions have been proposed to protect user privacy [**?**], [**?**]. Among them, Pairwise Pseudonymous Identifier (PPID) [**?**], [**?**] is a most straightforward and commonly accepted solution to defend against RP-based identity linkage, which requires the IdP to create different identifiers for the user when she logs in to different RPs. In this way, even multiple malicious RPs collude with each other across the system, they cannot link the pairwise pseudonymous identifiers of the user and track which RPs she has visited. 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 access tracing, as the IdP still knows which RP the user visits.

To the best of our knowledge, there are only two schemes (i.e., BrowserID [**?**] and SPRESSO [**?**]) being proposed so far to prevent IdP-based access tracing. In BrowserID (and its prototype system known as Mozilla Persona [**?**] and Firefox Accounts [**?**]), IdP generates a user certificate to bind the user's unique identifier (i.e., email address) with a public key; while the user will use the corresponding private key to bind the identity proof (including the user certificate) with an RP, and send it to the correct RP confidentially. In SPRESSO, the RP chooses a third-party entity (named forwarder) as the proxy to receive the identity proof, and generates a pseudonymous identifier for itself on each login; IdP generates the identity proof for the user, binds it with the RP's pseudonymous identifier and sends the encrypted proof to the forwarder; while, the forwarder transmits the identity proof to the correct RP who performs the decryption and obtains the plain-text identity proof. In these two schemes, without the identifiers of the visiting RPs, the IdP needs to include the user's unique identifier (e.g., email address) in the identity proof, which is necessary for each RP to obtain a common account during the user's multiple logins. Then, the collusive RPs could perform RP-based identity linkage with the user's unique identifier.

As described above, none of existing SSO systems could prevent both the RP-based identity linkage and IdP-based access tracing. Here, we analyze these two privacy leakage problems formally. Each user has one identifier at the IdP and each RP respectively, which is denoted as ID_U at the IdP and $Account$ at the RP. Each RP has one global unique identifier (denoted as ID_{RP}), and a privacy-preserving identifier PID_{RP} (may be null) at IdP for each login. PID_{RP} is generated by the RP or the user, with the function $PID_{RP} = F_{PID_{RP}}(ID_{RP})$. And, IdP generates a privacy-preserving user identifier (PID_U) with ID_U and PID_{RP}, based on the function $PID_U = F_{PID_U}(ID_U, PID_{RP})$. While, RP calculates $Account$ with the function $Accout = F_{Accout}(PID_U, ID_{RP}, PID_{RP})$. The three functions F_{PID_U}, $F_{PID_{RP}}$ and F_{Accout} have to satisfy:

- To prevent RP-based identity linkage, F_{PID_U} needs to ensure that PID_U are unlinkable among various RPs.
- To prevent IdP-based access tracing, $F_{PID_{RP}}$ needs to ensure that PID_{RP} are unlinkable for multiple logins at an RP.
- To ensure each RP obtains the unchanged $Account$ among the user's multiple logins, all the functions ($F_{PID_{RP}}$, F_{PID_U} and F_{Accout}) need to be designed corporately, and therefore the output of $F_{Accout}(F_{PID_U}(ID_U, F_{PID_{RP}}(ID_{RP})), ID_{RP})$ will be unchanged for a user's multiple logins at an RP.

Existing privacy-preserving schemes adopt either the unsatisfying F_{PID_U} or $F_{PID_{RP}}$, which will simplify the construction of F_{Accout}, but fail to provide the complete user privacy. For example, in PPID [**?**], [**?**], $F_{PID_{RP}}$ outputs ID_{RP} directly and F_{Accout} outputs PID_U as $Accout$, then IdP knows which RP the user visits; in BrowserID [**?**] and SPRESSO [**?**], F_{PID_U} outputs ID_U directly and F_{Accout} uses ID_U as $Accout$ directly, then the collusive RPs could perform RP-based identity linkage.

In this paper, we propose UPRESSO, an Unlinkable Privacy-REspecting Single Sign-On system, which provides **all** the **satisfying** F_{PID_U}, $F_{PID_{RP}}$ and F_{Accout} based on the discrete logarithm problem, and prevents both the RP-based identity linkage and IdP-based access tracing. In UPRESSO, for each login, a one-way trapdoor function $F_{PID_{RP}}$ is invoked with a randomly chosen trapdoor (t) to generate a random and unique PID_{RP} which is anonymously registered it at the IdP by the user; then IdP invokes a one-way function F_{PID_U} to generate PID_U with PID_{RP} and ID_U, and issues the identity proof; finally, RP checks the binding and integrity of the identity proof, and invokes F_{Accout} with PID_U, ID_{RP}, PID_{RP} and the trapdoor t to obtain the unchanged $Accout$ for the user at this RP. Therefore, based on the discrete logarithm problem, UPRESSO ensures: (1) when a user logs in to an RP, the RP can derive a unchanged $Accout$ from different PID_U, but fails to infer ID_U; (2) when a user logs in to different RPs, various PID_Us are generated and collusive RPs fail to link the user's multiple logins; and (3) when a RP is visited during multiple logins, random PID_{RP}s are generated and the curious IdP cannot infer ID_{RP} nor link these logins.

We have implemented a prototype of UPRESSO based on an open-source implementation of OIDC. UPRESSO inherits the security properties from OIDC by achieving the binding, integrity and confidentiality of identity proof, and only requires small modifications to add the three functions F_{PID_U}, $F_{PID_{RP}}$ and F_{Accout} for privacy protection. Therefore, unlike BrowserID and SPRESSO which are the non-trivial re-designs of the existing SSO systems, UPRESSO is compatible with

existing SSO systems, and doesn't require a completely new and comprehensive formal security analysis.

The main contributions of UPRESSO are as follows:

- We systematically analyze the privacy issues in SSO systems and propose a comprehensive protection solution to hide users' traces from both curious IdPs and collusive RPs, for the first time. We also provide a systematic analysis to show that UPRESSO achieves the same security level as existing SSO systems.
- We develop a prototype of UPRESSO that is compatible with OIDC and demonstrate its effectiveness and efficiency with experiment evaluations.

The rest of this paper is organized as follows. We introduce the background in Sections **??**, and the challenges with solutions briefly **??**. Section **??** and Section **??** describe the threat model and the design of UPRESSO. A systematical analysis is presented in Section **??**. We provide the implementation specifics and evaluation in Section **??**, then introduce the related works in Section **??**, and draw the conclusion finally.

II. Background and Preliminary

UPRESSO is compatible with OIDC, and achieves the privacy protection based on the discrete logarithm problem. Here, we provide a brief introduction on OIDC and the discrete logarithm problem.

A. OpenID Connect

OIDC [**?**] is an extension of OAuth 2.0 to support user authentication, and becomes one of the most prominent SSO authentication protocols. Same as other SSO protocols [**?**], OIDC involves three entities, i.e., *users*, *identity provider (IdP)*, and *relying parties (RPs)*. Both users and RPs have to register at the IdP, the users register at the IdP to create credentials and identifiers (e.g. ID_U), while each RP registers at the IdP with its endpoint information to create its unique identifier (e.g., ID_{RP}) and the corresponding credential. IdP is assumed to securely maintain the attributes of users and RPs. Then, in the SSO authentication sessions, each user is responsible to start a login request at an RP, redirect the messages between RP and IdP, and check the scope of user's attributes provided to the RP; IdP authenticates the user, sets the $PPID$ for the user ID_U at the RP ID_{RP}, constructs the identity proof with $PPID$, ID_{RP} and the user's attributes consented by the user, and finally transmits the identity proof to the RP's registered endpoint (e.g., URL); each RP constructs an identity proof request with its identifier and the requested scope of user's attributes, sends an identity proof request to the IdP through the user, and parses the received identity proof to authenticate and authorize the user. Usually, the redirection and checking at the user are handled by a user-controlled software, called *user agent* (e.g., browser).

Implicit flow of user login. OIDC supports three processes for the SSO authentication session, known as *implicit flow*, *authorization code flow* and *hybrid flow* (i.e., a mix-up of the previous two). In the implicit flow of OIDC, a token, known as *id token*, is introduced as the identity proof, which contains

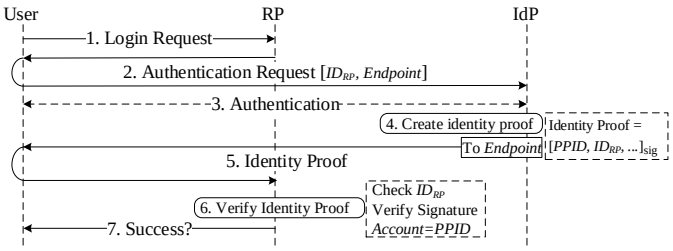

Fig. 1: The implicit protocol flow of OIDC.

user identifier (i.e., $PPID$), RP identifier (i.e., ID_{RP}), the issuer (i.e., IdP), issuing time, the validity period, and other requested attributes. The IdP signs the id token using its private key to ensure integrity, and sends it through the user to RP. In the authorization code flow, IdP binds an authorization code with the RP, and redirects this code to the RP; then RP establishes an HTTPS connection with IdP to ensure the integrity and confidentiality of the identity proof, and uses the authorization code with the RP's credential to obtain PPID and the user's other attributes.

UPRESSO is compatible to all the three flows. For brevity, we will present the application of UPRESSO in the implicit flow in details, and provide the integration with the authorization code flow briefly. Here, we first introduce the original processes in the implicit flow of OIDC. As shown in Figure **??**, the implicit flow of OIDC consists of 7 steps: when a user attempts to log in to an RP (Step 1), the RP constructs a request for identity proof, which is redirected by the user to the corresponding IdP (Step 2). The request contains ID_{RP}, RP's endpoint and a set of requested user attributes. If the user has not been authenticated yet, the IdP performs an authentication process (Step 3). If the RP's endpoint in the request matches the one registered at the IdP, it generates an identity proof (Step 4) and sends it back to the RP (Step 5). Otherwise, IdP generates a warning to notify the user about potential identity proof leakage. The RP verifies the id token (Step 6), extracts user identifier from the id token and returns the authentication result to the user (Step 7).

RP dynamic registration. OIDC provides a dynamic registration mechanism [**?**] for the RP to renew its ID_{RP} dynamically. When an RP first registers at the IdP, it obtains a registration token, with which the RP can invoke the dynamic registration process to update its information (e.g., the endpoint). After each successful dynamic registration, the RP obtains a new unique ID_{RP} from the IdP. In UPRESSO, we slightly modify dynamic registration to register PID_{RP} at IdP.

B. Discrete Logarithm Problem

Discrete logarithm problem is adopted in UPRESSO for the construction of $F_{PID_{RP}}$ and F_{PID_U}, which generate privacy-preserving user identifier (e.g. PID_U) and RP identifier (e.g. PID_{RP}) respectively. Here, we provide a brief description of the discrete logarithm problem.

A number g $(0 < g < p)$ is called a primitive root modular a prime p, if for $\forall y$ $(0 < y < p)$, there is a number x

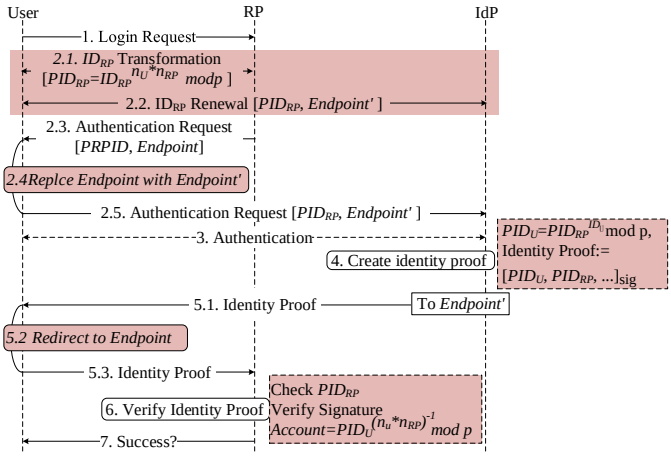

Fig. 2: The UPRESSO.

$(0 \leq x < p-1)$ satisfying $y = g^x \pmod p$. And, x is called the discrete logarithm of y modulo p. Given a large prime p, a primitive root g and a number y, it is computationally infeasible to derive the discrete logarithm (here x) of y (detailed in [**?**]), which is called discrete logarithm problem. The hardness of solving discrete logarithm has been used to construct several security primitives, including Diffie-Hellman key exchange and Digital Signature Algorithm (DSA).

In the process of $F_{PID_{RP}}$ and F_{PID_U}, we needs to calculate the primitive root for a large prime p as follows [**?**], [**?**]. First, we retrieve a primitive root g_m modulo p from all the integers by finding the first integer passing the primitive root checking. A lemma is propose to simply the checking, that if $p = 2q+1$ (q is a prime), an integer $\mu \in (1, p-1)$ is a primitive root if and only if $\mu^2 \neq 1 \ mod \ p$ and $\mu^q \neq 1 \ mod \ p$. Then, based on g_m, we can calculate a new primitive root $g = g_m^t mod \ p$, where t is an integer coprime to $p-1$.

III. The Privacy Dilemma in Single Sign-On

In this section, we describe the challenges for developing privacy-preserving SSO systems and provide an overview of the solutions proposed in UPRESSO.

A. Basic Security Requirements of SSO

Both the designs of SSO protocols and the implementations of SSO systems are challenging [**?**], and various vulnerabilities have been found in existing systems [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. With these vulnerabilities, the adversaries break the security of SSO systems and achieve the following two goals [**?**]:

- **Impersonation**: Adversary logs in to an honest RP as the victim user.
- **Identity injection**: A victim user logs in to an honest RP under the adversaries' identity.

We summarize basic requirements of SSO systems based on existing theoretical analysis [**?**], [**?**], [**?**] and practical attacks [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] on SSO systems. These basic requirements focus on functions and security of SSO systems, and are as follows:

User identification. When a user logs in to a same RP multiple times, the RP should be able to associate these logins to provide a continuous and personalized service to that user.

Receiver designation. The receiver designation requires that the identity proof should be transmitted only to the RP (and user) that the user visits, and be bound to this RP so that it will accepted only by this RP.

Integrity. Only the IdP is able to generate a valid identity proof, no other entity should be able to modify or forge it [**?**] without being found. And, the honest RP should only accept the valid identity proof.

These basic requirements are the minimum properties that an SSO system has to provide. User identification is necessary for all services except the anonymous systems which will be discussed in Section **??**, therefore RPs should be able to identify the user with the help from IdP. While either the receiver designation and integrity not satisfied, the impersonation and identity injection will exist in the SSO systems. For example, without the receiver designation, the adversaries cloud use the leaked identity proof to impersonate the victim user at an honest RP [**?**], [**?**], [**?**], and make the victim RP incorrectly accept the identity proof intended for other RP, which allows the adversary to perform impersonation directly or inject the identity proof in the web session between the victim user and honest RP with other web attacks (e.g., CSRF); while without integrity, the impersonation and identity injection attacks will be more easier as an adversary could directly modify user's identifier in the identity proof.

B. The Privacy Dilemma and Existing Attempts

In addition to the basic requirements, a privacy-preserving SSO system should prevent the IdP-based access tracing and RP-based identity linkage. In details, the privacy-preserving SSO system should prevent the curious IdP from obtaining any information that could identify or link the user's accessed RP (for example, through RP's identifier and URL), and prevent the collusive RPs from correlating the user's identifiers at different RPs.

The privacy-preserving requirements need to be integrated into the basic requirements, to protect the user privacy under the prerequisite that the correct functions and security of SSO systems are ensured. The privacy-preserving user identification requires IdP to provide a user's identifier (PID_U) to help the RP in identifying the user locally, under the prerequisites that the curious IdP can never identify or link the visiting RP and the collusive RPs cannot find the correlation between PID_Us for a user. The privacy-preserving receiver designation requires IdP (or the user) to bind an identity proof with an RP and send it only to this RP, while IdP can never identify or link the RP visited by the user.

The above analysis demonstrates that the identifers of the user and RP need to be carefully processed in SSO systems. Here, we systematically analyze the forms of the user and RP

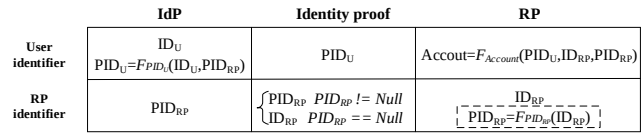

Fig. 3: Identifiers of a user and RP in a privacy-preserving SSO system.

identifiers, as required by privacy protection. IdP, who knows the user's globally unique identifer (ID_U), should only obtain a privacy-preserving identifer for an RP (PID_{RP}); each RP, who knows its globally unique identifer (ID_{RP}), should only receive a privacy-preserving identifer for a user (PID_U) from the identity proof. The PID_{RP} may be null, it means IdP has no information about RP, for example in BrowserID [**?**]. The PID_U can never be null, as RP has to derive the $Accout$ from it, which is the basic function of SSO systems. Here the privacy-preserving identifers in multiple sessions will never leak the globally unique identifer nor link the sessions from an entity (the RP and user). That is, PID_{RP}s in multiple sessions for a same RP should be independent from the view of the IdP; and for the collusive RPs, their obtained PID_Us for a same user should be independent.

Then, we analyze the generation and use of PID_U and PID_{RP} as in Figure **??**, considering the basic requirements of SSO system.

- IdP is the only trusted entity to control the generation of PID_U, as the user is not trusted by the RP and a malicious user may provide incorrect PID_U. Here, the "control" means, IdP may generate PID_U alone or with the corporation from the user, but it's the IdP who finally determines the value of PID_U. For clarity, we assume IdP generates PID_U with the function $F_{PID_U}(ID_U, PID_{RP})$, without loss of generality.
- The generation of PID_U and PID_{RP} must ensure the **user identification**, that is, the RP could derive a same $Account$ with the PID_U and PID_{RP} from different logins. We assume RP calculates $Account$ with the function $F_{Account}(PID_U, ID_{RP}, PID_{RP})$.
- Each PID_{RP} must be globally unique, i.e., only assigned to one RP, for achieving the **receiver designation**. The user and RP may generate the PID_{RP} separately or cooperatively. However, both the user and RP must check the uniqueness of PID_{RP}. If either the user or the RP doesn't perform the check, the adversary could make it accept a PID_{RP} same as an RP and then misuse the identity proof. PID_{RP} is one form of the RP's identifer, and seems unrelated with the user's identifier. Although, the user may inject the user's information into PID_{RP}, these information will be treated as random values at the RP and never be used to calculate $Account$, as the user is not trusted by the RP. Therefore, we can assume that PID_{RP} is generated through a function $F_{PID_{RP}}(ID_{RP})$. [1]
- The **receiver designation** further requires that PID_U is bound with either a non-null PID_{RP} or ID_{RP} in identity proof. When PID_{RP} is non-null, IdP builds the identity proof separately and the **integrity** is also ensured. When PID_{RP} is null, only the user cloud bind PID_U with an RP identifier (i.e., ID_{RP}) which is unique and checkable to the RP. In this case, the user who performs the binding, must have a publicly verifiable grant from the IdP, as required by **integrity**. The binding and integrity cloud be achieved by existing public key infrastructure.
- The **receiver designation** also requires the identity proof will only be sent to the correct RP. As IdP doesn't know ID_{RP}, the user or a third party trusted by the correct RP will ensure this.

Then, the dilemma in the design of a secure and privacy-preserving SSO system, could be transformed into finding the three functions $F_{PID_{RP}}$, F_{PID_U} and $F_{Account}$, which satisfying:

- For an RP, $F_{PID_{RP}}$ generates PID_{RP}s in multiple logins, and these PID_{RP}s are independent to the IdP.
- For a user, F_{PID_U} generates PID_Us in multiple logins at different RPs, and these PID_Us are independent to these RPs.
- For a user and an RP, $F_{Account}$ generates a unchanged $Account$ in multiple logins.

Various solutions [**?**], [**?**], [**?**], [**?**], are proposed, attempting to construct a secure and privacy-preserving SSO system. However, these scheme provide at most two satisfying functions, and therefore fail to prevent either the IdP-based access tracing or RP-based identity linkage.

- The traditional SSO systems provide only the satisfying $F_{Account}$, and therefore fail to protect the user's privacy.
- SAML [**?**] and OIDC [**?**] provide only the satisfying F_{PID_U} and $F_{Account}$. The IdP obtains the ID_{RP} for an RP, and generates the unchanged PID_U for the same couple $< ID_U, ID_{RP} >$, while the PID_U are independent for different ID_{RP}s.
- BrowserID [**?**] and SPRESSO [**?**] provide only the satisfying $F_{PID_{RP}}$ and $F_{Account}$. In BrowserID, IdP obtains a null PID_{RP} and provides ID_U to the RP, therefore each RP obtains the unchanged $Accout$. In SPRESSO, each RP generates PID_{RP} by encrypting ID_{RP} with a random nonce, and IdP provides the unchanged ID_U for a user's multiple logins no matter which RP the user is visiting.

Obliviously, existing attempts fail to provide the complete privacy. The essential reason is that these schemes provide an unchanged value (e.g., PID_U in SAML [**?**] and OIDC [**?**], or ID_U in BrowserID [**?**] and SPRESSO [**?**]) for the user's multiple logins at an RP. To provide unchanged PID_U, IdP will be able to identity or link the logins at an RP. Providing

[1]The PID_{RP} may also be related with the previous PID_{RP}. We omit the previous PID_{RP} in the equation as it is also a transformation of ID_{RP}.

ID_U to the RP, makes the collusive RPs easily link the user's logins at different RPs.

C. The Principles of UPRESSO

In this work, we present UPRESSO, a secure and privacy-preserving SSO system, which prevents the IdP-based access tracing and RP-based identity linkage under the prerequisites that the basic requirements of SSO systems are satisfied. UPRESSO adopts the public key infrastructure to ensure the integrity of the identity proof, which is the same as other SSO systems. Therefore, we focus on how to provide the privacy-preserving user identification and receiver designation as follows:

Trapdoor user identification. UPRESSO breaks the implicit assumption in previous SSO systems, that an RP should obtain an unchanged value to identify a user in the multiple logins. A trapdoor identification is introduced, which allows the RP with a trapdoor to derive the unchanged $Accout$ from different PID_Us. The trapdoor identification requires a cooperation of the three functions $F_{PID_{RP}}$, F_{PID_U} and $F_{Account}$. $F_{PID_{RP}}$ is invoked with a trapdoor to generate $PID_{RP}s$ which are independent to IdP (preventing IdP-based access tracing), and RP uses $F_{Account}$ with this trapdoor to derive the unchanged $Accout$ from PID_Us that are generated by IdP with F_{PID_U}.

Transformed receiver designation. UPRESSO splits the receiver designation into two steps: IdP designates the identity proof to a transformed RP identifer (i.e., PID_{RP}), and the user designates the PID_{RP} to the correct ID_{RP}. Each RP checks the designation of the identity proof based on PID_{RP}. And, the designation between PID_{RP} and ID_{RP} ensures that PID_{RP} is correct and the identity proof is only sent to the correct RP. In the challenging first step, the IdP generates PID_U for PID_{RP} and achieves full privacy-preserving binding (i.e., PID_U with PID_{RP}). UPRESSO introduces an efficient one-way (trapdoor) function $F_{PID_U}(ID_U, PID_{RP})$. It allows IdP to compute PID_U easily, avoiding the generation of PID_U to be the bottleneck at a high-throughput IdP; and also prevents the RP from finding any information about ID_U, which is required by preventing RP-based identity linkage. As IdP only knows PID_{RP} and has no other information about RP, the user is required to perform the necessary checks in the second step. The user will check that PID_{RP} is global unique and corresponding to the correct RP. Finally, the user sends the identity proof to the only correct RP.

To meet the above two principles, we need to construct three satisfying functions $F_{PID_{RP}}$, F_{PID_U} and $F_{Account}$, design the protocols between the user, RP and IdP to avoid the privacy leakage during message transmission, and implement the processing at the user as required by the transformed receiver designation.

IV. Threat Model and Assumption

To be compatible the traditional SSO systems (e.g., SAML, OIDC), UPRESSO doesn't introduce any other entity, but only modifies the processes at existing entities, i.e, one IdP, multiple RPs and users, to provide the secure and privacy-preserving SSO service. Here, we introduce the threat model and assumptions in UPRESSO.

A. Threat Model

In UPRESSO, the IdP is assumed to be semi-honest, while the users and RPs could be controlled by the adversary and be malicious. The malicious users and RPs could behave arbitrarily and collude with each other for breaking the security and privacy of correct users. While, the IdP will follow the protocol correctly, and is only curious about the user's privacy. The details are as follows.

Semi-honest IdP. We assume the IdP is well-protected and will never leak any sensitive information. For example, the private key for generating the identity proof and RP certificate (used in Section **??**) will never be leaked, therefore the adversary fails to impersonate as the IdP to forge a valid identity proof or RP certificate. The honest IdP processes the requests of RP registration and identity proof correctly, and never colludes with others (e.g., malicious RPs and users). For example, IdP ensures the uniqueness of ID_{RP} and PID_{RP}, and generates the correct RP certificate, PID_U and identity proof. However, the curious IdP may attempt to break the user's privacy without violating the protocol. For example, the curious IdP may store and analyze the received messages, and perform the timing attacks, attempting to achieve the IdP-based linkage.

Malicious users. The adversary could control a set of users, for example through stealing the users' credentials [**?**] or registering at the IdP and RPs directly. These malicious users aim to break the security of the SSO system. That is, they attempt to impersonate an uncontrolled user at the victim RP, and make a victim user log in at the correct RP under a controlled identity. To achieve this, they could behave arbitrarily [**?**], [**?**]. For example, the malicious users may forge the identity proof, modify the forwarding messages (requests of identity proof, identity proof, RP registration request and result, and etc.), and provide incorrect values for negotiating PID_{RP} (detailed in Section **??**).

Malicious RPs. The adversary could control a set of RPs, by registering an RP at the IdP or exploiting various vulneraries to attack RPs. These malicious RPs aim to break the security and privacy of the correct users, and could behave arbitrarily. For example, to break the security, the malicious RPs need to obtain an identity proof valid for other RP, and attempt to achieve this by behaving as follows: impersonating other RP at the user by providing the incorrect RP certificate, using incorrect values during the negotiation of PID_{RP} to make the generated PID_{RP} be same as the one for other RP, or constructing an incorrect request to trigger the IdP issuing an identity proof binding with other RP. Moreover, the malicious RPs may attempt to perform the RP-based identity linkage and break the user's privacy. To achieve this, the RPs could behave arbitrarily and collude with each other. For example, the RPs may attempt to derive the ID_U from PID_U by

providing incorrect values to the IdP, and the colluded RPs may attempt to link the user's multiple logins, by providing correlated values (e.g., PID_{RP}) to the IdP.

Collusive users and RPs. The malicious users and RPs may collude and behave arbitrarily, attempting to break the security of UPRESSO. For example, the adversary may first act as a malicious RP, and make an incorrect identity proof generated for the visiting user, then act a malicious user, and use this identity proof to impersonate this victim user at another RP. The adversary could also first act as a user to login a correct RP and obtain an identity proof, then act a malicious RP to perform the identity injection attack, by injecting this identity proof to the session between the victim user and the correct RP with other web attacks (e.g., CSRF).

B. Assumption

In UPRESSO, we assume that the user agent deployed at the honest user is correctly implemented, and will transmit the messages to the correct destination. The TLS is also correctly implemented at the user agent, IdP and RP, which ensures the confidentiality and integrity of the network traffic between correct entities. We also assume a secure random number generator is adopted in UPRESSO to provide the unpredictable random numbers; and the adopted cryptographic algorithms, including the RSA and SHA-256, are secure and implemented correctly. Therefore, no one without private key can forge the signature, and the adversary fails to infer the private key during the computation. Moreover, we also assume the security of the discrete logarithm problem is ensured.

The collusive RPs may attempt to link a user based on the identifying attributes, such as the telephone number and credit number. Here, we assume that the users refuse to provide these attributes to the RPs, and the correct RPs never collect these attributes as required by privacy laws (e.g., GDPR). Moreover, the global network traffic analysis may be adopted to correlate the user's logins at different RPs. However, UPRESSO may integrate existing defenses to prevent this attack.

V. Design of UPRESSO

In this section, we first present the features of UPRESSO to confirm the requirement of principles trapdoor user identification and transformed receiver designation. Then, we describe the implementations for these features , containing system initialization, initial registration of RP and the description of algorithms for calculating the RP identifier transformation, user identifier and account. The detailed processing for each user's login is provided corresponding to the main process phases. Finally, we discuss the compatibility of UPRESSO with OIDC.

A. Features

In Section **??**, we highlight the design principles of UPRESSO, namely trapdoor user identification and transformed receiver designation, to develop a secure and privacy-preserving SSO. Next, we will discuss the features to confirm these requirements. To ease the presentation, we list the notations used in this paper in Table **??**.

Using the novel one-time privacy-preserving RP identifier generating method for acquiring trapdoor and pseudonymous RP identifier in each authentication. The novel RP identifier generation is based on the negotiation between RP and user to prevent potential attacks, which is to be described in Section **??**. This privacy-preserving RP identifier (denoted as PID_{RP}) is generated from the original identifier of the RP (denoted as ID_{RP}) through a transformation negotiated between a pair of cooperating RP and user in the *RP identifier transforming* phase (Step 2.1 in Figure **??**). Moreover, during ID_{RP} negotiation, RP also acquires the trapdoor (denoted as t), which is to be used for further user identification.

Using the novel RP-identifier-based user identifier generating method for authenticating the user through trapdoor. IdP should generate the unique privacy-preserving user identifier based on the original user identifier and the privacy-preserving RP identifier in a way that a same user account can be derived through the trapdoor from multiple privacy-preserving user identifiers for the same user at same RP.

Using user-centric verification for correct identity proof generating and transmitting. The user agent should guarantee that the identity proof is only generated based on the legal ID_{RP} and sent to the corresponding RP by checking the binding between ID_{RP} and its endpoint. It requires that the IdP should provide the unforgeable proof for valid ID_{RP} and endpoint.

B. The implementations of UPRESSO

To lead the forementioned features be implemented in UPRESSO, it requires the system initialization is performed to initialize the IdP only once at the very beginning of system construction and generates key parameters for further initiation and login process. Moreover, the RP should be initiated before it attaches the UPRESSO system, when the IdP provides the ID_{RP} and unforgeable RP proof (denoted as $Cert_{RP}$). Next, we will describe the System initialization, RP initial registration and the algorithms for calculating PID_{RP}, PID_U and $Account$.

System initialization. The IdP generates the random asymmetric key pair, (SK, PK), for calculating the signatures in the identity proof and $Cert_{RP}$, respectively; and provides PK and PK as the public parameters for the verification of identity proof and $Cert_{RP}$. Moreover, IdP generates a strong prime p, calculates a primitive root (g), and provides p and g as the public parameters. For p, we firstly randomly choose a large prime q, and accept $2q+1$ as p if $2q+1$ is a prime. The strong prime p makes it easier to choose n_u and n_{RP} at the user and RP for the RP identifier transformation (detailed in Section **??**). Once SK leaked, IdP may update this asymmetric key pair. However, g and p will never be modified, otherwise, the RPs fail to link the user's accounts between two different p (or g).

TABLE I: The notations used in UPRESSO.

Notation	Definition	Attribute
p	A large prime.	System-unique
g	A primitive root modulo p.	System-unique
$Cert_{RP}$	An RP certificate.	System-unique
SK, PK	The private/public key of IdP.	System-unique
ID_U	User's unique identifier at IdP.	System-unique
PID_U	User's privacy-preserving id in the identity proof.	One-time
$Account$	User's identifier at an RP.	RP-unique
ID_{RP}	RP's original identifier.	System-unique
PID_{RP}	The privacy-preserving ID_{RP} transformation.	One-time
n_U	User-generated random nonce for PID_{RP}.	One-time
n_{RP}	RP-generated random nonce for PID_{RP}.	One-time
Y_{RP}	Public value for n_{RP}, $(ID_{RP})^{n_{RP}}\ modp$.	One-time
t	A trapdoor, $t = (n_U * n_{RP})^{-1} mod\ (p-1)$.	One-time

RP initial registration. The RP invokes an initial registration to apply a valid and self-verifying $Cert_{RP}$ from IdP (**Goal 1**), which contains three steps:

1) RP sends IdP a $Cert_{RP}$ request $Req_{Cert_{RP}}$, which contains the distinguished name $Name_{RP}$ (e.g., DNS name) and the endpoint to receive the identity proof.
2) IdP calculates $ID_{RP} = g^r mod\ p$ with a random chosen r which is coprime to $p-1$ and different from the ones for other RPs, generates the signature (Sig_{SK}) of $[ID_{RP}, Name_{RP}]$ with SK, and returns $[ID_{RP}, Name_{RP}, Sig_{SK}]$ as $Cert_{RP}$.
3) IdP sends $Cert_{RP}$ to the RP who verifies $Cert_{RP}$ using PK.

To satisfy RP fails to derive ID_U from $Account$ and the $Accounts$ of a user are different among RPs in **Goal 3**, two requirements on r are needed in ID_{RP} generation:

- r should be coprime to $p-1$. This makes ID_{RP} also be a primitive root, and then prevents the RP from inferring ID_U from $Account$, which is ensured by the discrete logarithm problem.
- r should be different for different RPs. Otherwise, the RPs assigned the same r (i.e., ID_{RP}), will derive the same $Account$ for a user, resulting in identity linkage.

Algorithms for calculating PID_{RP}, PID_U and $Account$. Moreover, we provide the algorithms for calculating PID_{RP}, PID_U and $Account$, which are the foundations to process each user's login.

PID_{RP}. Similar to Diffie-Hellman key exchange [**?**], the RP and user negotiate the PID_{RP} as follows:

- RP chooses a random odd number n_{RP}, and sends $Y_{RP} = {ID_{RP}}^{n_{RP}} mod\ p$ to the user.
- The user replies a random chosen odd number n_u to the RP, and calculates $PID_{RP} = {Y_{RP}}^{n_u} mod\ p$.
- RP also derives PID_{RP} with the received n_u.

As denoted in Equation **??**, ID_{RP} cannot be derived from PID_{RP}, which satisfies IdP never infers ID_{RP} from PID_{RP} in **Goal 2**.

$$PID_{RP} = {Y_{RP}}^{n_u} = {ID_{RP}}^{n_u * n_{RP}} mod\ p \tag{1}$$

To ensure PID_{RP} not controlled by the adversary in **Goal 2**, PID_{RP} should not be determined by either the RP or user. Otherwise, malicious users may make the victim RP accept an identity proof issued for another RP; or collusive RPs may provide the correlated PID_{RP} for potential identity linkage. In UPRESSO, RP fails to control the PID_{RP} generation, as it provides Y_{RP} before obtaining n_u and the modification of Y_{RP} will trigger the user to generate another different n_u. The Discrete Logarithm problem prevents the user from choosing an n_u for a specified PID_{RP} on the received Y_{RP}.

In UPRESSO, both n_{RP} and n_u are odd numbers, therefore $n_{RP} * n_u$ is an odd number and coprime to the even $p-1$, which ensures that the inverse $(n_{RP} * n_u)^{-1}$ always exists, where $(n_{RP} * n_u)^{-1} * (n_{RP} * n_u) = 1\ mod\ (p-1)$. The inverse serves as the trapdoor t for $Accout$, which makes:

$$(PID_{RP})^t = ID_{RP}\ mod\ p \tag{2}$$

PID_U. The IdP generates the PID_U based on the user's ID_U and the user-provided PID_{RP}, as denoted in Equation **??**. The corporative generation of PID_{RP}, ensures unrelated PID_U for different RPs in **Goal 3**. Also, RP fails to derive ID_U from either PID_U in **Goal 3**, as PID_{RP} is a primite root modulo p, and the ID_U cannot be derived from $PID_{RP}^{ID_U}$.

$$PID_U = {PID_{RP}}^{ID_U}\ mod\ p \tag{3}$$

Finally, the RP calculates $PID_U^t mod\ p$ as the user's account, where PID_U is received from the user and t is derived in the generation of PID_{RP}. Equation **??** demonstrates that $Account$ is unchanged to the RP during a user's multiple logins, satisfying RP derives the user's unique $Account$ from PID_U with the trapdoor and RP fails to derive ID_U from either $Account$ in **Goal 3**. The different ID_{RP} ensures the $Accounts$ of a user are different among RPs in **Goal 3**.

$$Account = ({PID_{RP}}^{ID_U})^t = {ID_{RP}}^{ID_U} mod\ p \tag{4}$$

C. Overall protocol flow overview

In this section, we present the detailed process for each user's login as shown in Figure **??**.

In RP identifer transforming phase, the user and RP corporately process as follows. **(1)** The user sends a login request to trigger the negotiation of PID_{RP}. **(2.1.1)** RP chooses the random n_{RP}, and calculates Y_{RP} as described in Section **??**.

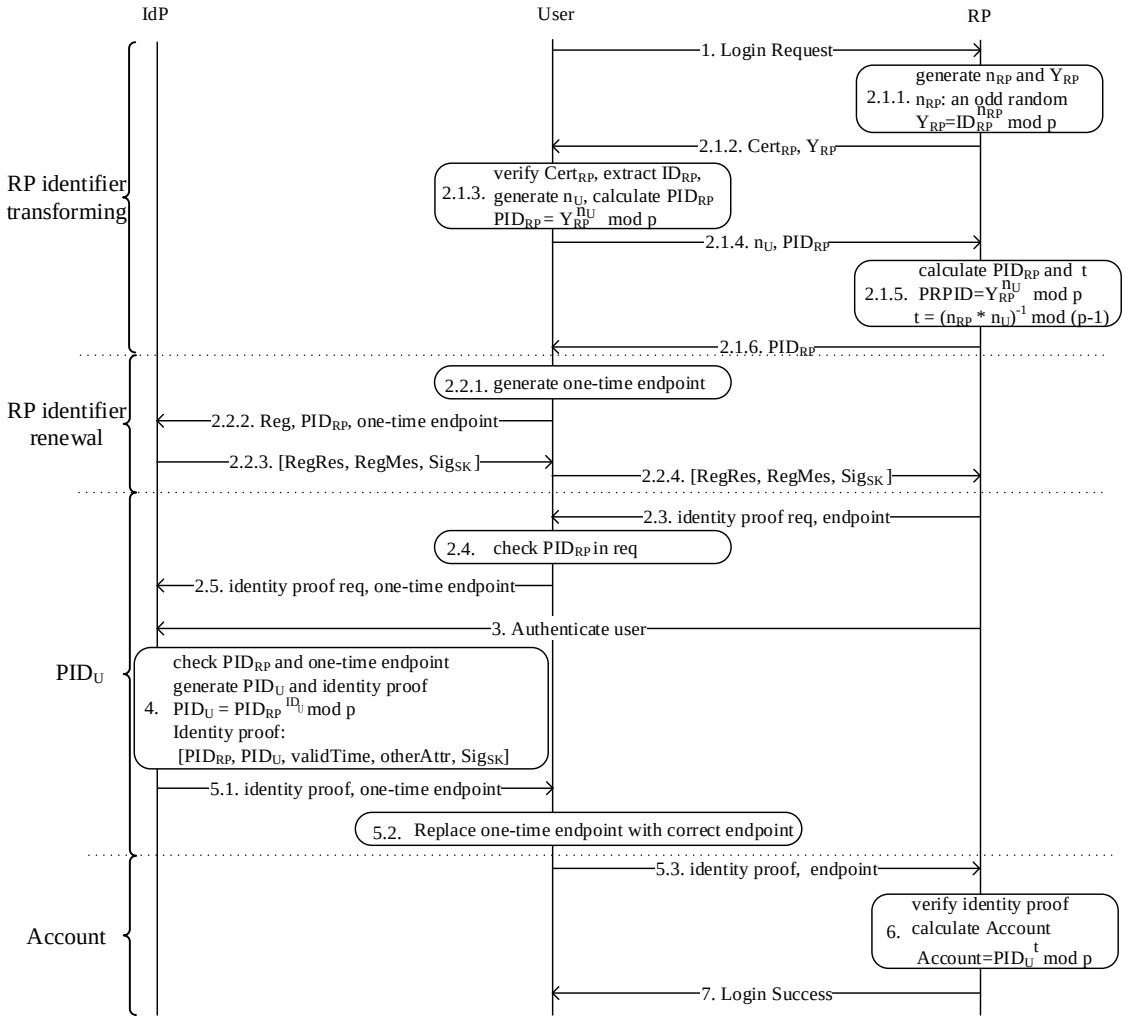

Fig. 4: Process for each user login.

(2.1.2) RP sends $Cert_{RP}$ with Y_{RP} to the user. **(2.1.3)** The user halts the login process if the provided $Cert_{RP}$ is invalid; otherwise, it extracts ID_{RP} from $Cert_{RP}$, and calculates PID_{RP} with a random chosen n_U as in Section **??**. **(2.1.4)** The user sends n_U to the RP. **(2.1.5)** RP calculates PID_{RP} using the received n_U with Y_{RP} as in Section **??**. After that, RP derives the trapdoor t as in Section **??**, which will be used in calculating $Account$. **(2.1.6)** RP sends the calculated PID_{RP} to the user, who will halt the login if the received PID_{RP} is different from the cached one.

In the RP identifier renewal phase, the user registers the newly generated PID_{RP} at the IdP as follows. **(2.2.1)** The user generates an one-time endpoint (used in Section **??**) if the received PID_{RP} is accepted. **(2.2.2)** Then, the user registers the RP with the PID_{RP} and one-time endpoint. **(2.2.3)** If PID_{RP} is globally unique and is a primitive root module p, IdP sets the flag $RegRes$ as OK (otherwise $FAIL$), and constructs the reply in the form of [$RegRes$, $RegMes$, Sig_{SK}] where $RegMes$ is the response to original dynamic registration containing PID_{RP}, issuing time as well as other attributes and Sig_{SK} is the signature of the other elements using the private key SK_{ID} (ensuring unique PID_{RP} for binding in **Goal 2**). **(2.2.4)** The user forwards the registration result to the RP. The user obtains $RegRes$ directly as the connection between the user and IdP is secure, while the RP accepts the $RegRes$ only when Sig_{SK} is valid and $RegMes$ is issued for the PID_{RP} within the valid period. The user and RP will negotiate a new PID_{RP} if $RegRes$ is $FAIL$.

To acquire the PID_U, the user corporates with the RP and IdP as follows. **(2.3)** RP constructs an identity proof request with the correctly registered PID_{RP} and the endpoint (the form of the request is detailed in Section **??**). **(2.4)** The user halts the login process if the received PID_{RP} is different from the previous one. **(2.5)** The user replaces the endpoint with the registered one-time endpoint, and sends it with the identity proof request to the IdP. **(3)** IdP requires the user to provide the correct credentials if the user hasn't been unauthenticated. **(4)** IdP rejects the request if the binding of PID_{RP} and the one-time endpoint doesn't exist in the registered ones. Then, IdP generates the PID_U as in Section **??**, and constructs the

identity proof with PID_{RP}, PID_U, the valid period, issuing time and other attribute values, by attaching a signature of these elements using the private key SK. **(5.1)** IdP sends the identity proof with the one-time endpoint to the user. **(5.2)** User agent replaces the one-time endpoint with the correct endpoint. **(5.3)** The user forwards the identity proof to the RP's endpoint corresponding to the one-time endpoint.

Finally, RP derives the user's $Account$ from PID_U as follows. **(6)** RP accepts the identity proof only when the signature is correctly verified with PK, PID_{RP} is the same as the negotiated one, the issuing time is less than current time, and the current time is in the validity period. If the identity proof is incorrect, RP returns login fail to the user who will trigger another login request. Otherwise, RP calculates the $Account$ as in Section **??**. **(7)** RP sends the login result to the user and begins to provide the personalized service.

D. Compatibility with OIDC

UPRESSO is compatible with the implicit protocol flow of OIDC (authorization code flow is discussed in Section **??**).

Consistent with OIDC. The entities in UPRESSO are same as them in OIDC (e.g. IdP, RP and user), which means it is not needed to introduce new trusted parties into this system and able to make UPRESSO owns the similar process with OIDC. The RP identifier renewal, PID_U generation and $Account$ acquiring phases are in line with the OIDC process.

In UPRESSO, the formats of identity proof request and identity proof are the same as the ones in OIDC. In details, each element of the identity proof request in OIDC is contained in UPRESSO as follows: the RP's identifier (PID_{RP} in UPRESSO), the endpoint (one-time endpoint in the request from the user in UPRESSO) and the set of required attributes (also supported by UPRESSO but not listed here). The identity proof in UPRESSO is also exactly the same as the one in OIDC, which includes RP's identifer (PID_{RP} in UPRESSO), the user's PPID (PID_U in UPRESSO), the issuer, validity period, issuing time, other requested attributes and a signature generated by SK_{IdP}.

The same formats of identity proof request and identity proof make the verification same in OIDC and UPRESSO. The IdP, in both UPRESSO and OIDC, verifies the identity proof request, by checking whether the mapping of RP's identifier and endpoint exists in the registered ones. The RP, in both UPRESSO and OIDC, verifies the identity proof, by checking the signature, the consistency of RP's identifer in the identity proof and the one it owns, the validity period, issuing time and the freshness (which is checked based on a nonce in OIDC, while PID_{RP} serves as the nonce in UPRESSO).

To avoid extra processes are added to UPRESSO compared with OIDC, the RP identifier renewal is implemented based on the dynamic registration (described in Section **??**) provided by IdP in OIDC.

Minimal modification to OIDC. Although the process is same in UPRESSO and OIDC, there is also minimal modification in the values generating and parsing.

In UPRESSO, the dynamic registration in RP identifier renewal is invoked by the user instead of the RP to prevent the curious IdP linking the new registered RP identifier with specific RP. As the registration response is transmitted through the user instead of server-to-server transmission between RP and IdP, the extra signature is required to guarantee the integrity of response.

In the PID_U generation phase, compared with original $PPID$ picking in OIDC (IdP pick the corresponding $PPID$ from database based on the RP and user identifier), the PID_U is temporarily calculated with current PID_{RP} and ID_U for each login instead of previously generated (for the first request) for all authentication requesting (expect for the first request).

In $Account$ acquiring phase, RP need the extra process to derive the user RP-uniquely identifier $Account$ from PID_U which does not exist in OIDC as the $PPID$ is able take the same responsibility as $Account$.

Newly added features. Compared with OIDC, the RP identifier transforming phase is added in UPRESSO to build one-time temporary transformed RP identifier (e.g., PID_{RP}). The RP identifier transforming required the negotiation between RP and user to generated the random identifier based on RP's registered identifier (e.g., ID_{RP}) while no entities are able to solely decide the result of negotiation.

Moreover, the negotiation conducted by user in RP identifier transforming phase, the one-time endpoint generating and storing in RP identifier renewal phase and the PID_{RP} checking in PID_U generation phase depends on the trusted user agent requires the more functional user agent in UPRESSO.

VI. Analysis

In this section, we firstly prove the privacy of UPRESSO, i.e., avoiding the identity linkage at the collusive malicious RPs, and preventing the curious IdP from inferring the user's accessed RPs. Then, we prove that UPRESSO does not degrade the security of SSO systems by comparing it with OIDC, which has been formally analyzed in [**?**].

A. Security

UPRESSO protects the user's privacy without breaking the security. That is, UPRESSO still prevents the malicious RPs and users from breaking the user identification, receiver designation and integrity.

In UPRESSO, all mechanisms for integrity are inherited from OIDC. The IdP uses the un-leaked private key SK_{ID} to prevent the forging and modification of identity proof. The honest RP (i.e., the target of the adversary) checks the signature using the public key PK_{ID}, and only accepts the elements protected by the signature.

For the requirement of receiver designation of identity proof, UPRESSO inherits the same idea (trusted transmission of identity proof and binding the identity proof with specific RP) from OIDC.

For example, in UPRESSO TLS, a trusted user agent and the checks are also adopted to guarantee the trusted transmission.

TLS avoids the leakage and modification during the transmitting. The trusted agent ensures the identity proof to be sent to the correct RP based on the endpoint specified in the $Cert_{RP}$. The $Cert_{RP}$ is protected by the signature with the un-leaked private key SK_{Cert}, ensuring it will never be tampered with by the the adversary. For UPRESSO, the check at RP's information is exactly the same as OIDC, that is, checking the RP identifier and endpoint in the identity proof request with the registered ones, preventing the adversary from triggering the IdP to generate an incorrect proof and transmitting it to the incorrect RP. However, the user in UPRESSO performs a two-step check instead of the direct check based on the ID_{RP} in OIDC. Firstly, the user checks the correctness of $Cert_{RP}$ and extracts ID_{RP} and the endpoint. In the second step, the user checks whether the RP identifier in identity proof request is the PID_{RP} negotiated for this authentication based on the ID_{RP} and the endpoint is also the one in $Cert_{RP}$. This two-step check also ensures the identity proof for the correct RP (ID_{RP}) is sent to correct endpoint (one specified in $Cert_{RP}$).

The mechanisms for binding are also inherited from OIDC. The IdP binds the identity proof with PID_{RP}, and the correct RP checks the binding by comparing the PID_{RP} with the cached one.

Receiver designation. UPRESSO binds the identity proof with PID_{RP}, instead of IdP chosen RP identifier for each RP assigned by IdP in OIDC. However, the adversary (malicious users and RPs) still fails to make one identity proof accepted by another honest RP. As the honest RP only accepts the valid identity proof for its fresh negotiated PID_{RP}, we only need to ensure one PID_{RP} (or its transformation) never be accepted by the other honest RPs.

- PID_{RP} is unique in one IdP. The honest IdP checks the uniqueness of PID_{RP} in its scope during the dynamic registration, to avoid one PID_{RP} (in its generated identity proof) corresponding to two or more RPs. Otherwise, there is hardly to be the conflicts between different login flows. We assume that the dynamically registered RP identifier is valid in 5 minutes (also the valid ticket window) and there are 1 billion (2^{30}) login requests during this period, as the chosen prime number p is 2048-bit long, the probability of conflict existing is about $1 - \prod_{i=1}^{2^{30}}(2^{2048-i}/2^{2048})$, which is almost 0. Moreover, the mapping of PID_{RP} and IdP globally unique. The identity proof contains the identifier of IdP (i.e., `issuer`), which is checked by the correct RPs. Therefore, the same PID_{RP} in different IdPs will be distinguished.
- The correct RP or user prevents the adversary from manipulating the PID_{RP}. For extra benefits, the adversary can only know or control one entity in the login flow (if controlling the two ends, no victim exists). The other honest entity provides a random nonce (n_U or n_{RP}) for PID_{RP}. The nonce is independent from the ones previously generated by itself and the ones generated by others, which prevents the adversary from controlling the PID_{RP}. Moreover, The PID_{RP} in the identity proof is protected by the signature generated with SK_{ID}. The adversary fails to replace it with a transformation without invaliding the signature.

User identification. UPRESSO ensures the identification by binding the identity proof with PID_U in the form of $PID_{RP}^{ID_U}$, instead of a randomly IdP generated unique identifier. However, the adversary still fails to login at the honest RP using a same $Account$ as the honest user. Firstly, the adversary fails to modify the PID_U directly in the identity protected by SK_{ID}. Secondly, the malicious users and RPs fail to trigger the IdP generate a wanted PID_U, as they cannot (1) obtain the honest user's PID_U at the honest RP; (2) infer the ID_U of any user from all the received information (e.g., PID_U) and the calculated ones (e.g., $Account$); and (3) control the PID_{RP} with the participation of a correct user or RP.

Protection conducted by user agent. The design of UPRESSO makes it immune to some existing known attacks [**?**] (e.g., CSRF, 307 Redirect) on the implementations. The Cross-Site Request Forgery (CSRF) attack is usually exploited by the adversary to perform the identity injection. However, in UPRESSO, the honest user logs PID_{RP} and one-time endpoint in the session, and performs the checks before sending the identity proof to the RP's endpoint, which prevents the CSRF attack. The 307 Redirect attacks [**?**] is due to the implementation error at the IdP, i.e. returning the incorrect status code (i.e., 307), which makes the IdP leak the user's credential to the RPs during the redirection. In UPRESSO, the redirection is intercepted by the trusted user agent which removes these sensitive information.

B. Privacy

Curious IdP. In the SSO schemes that do not protect user's privacy form IdP, e.g. OIDC, IdP is able to know the user accessed RP directly from the RP identifier (known as $clientid$). However, it fails to obtain the user's accessed RPs directly in UPRESSO. The curious IdP always fails to derive RP's identifying information (i.e., ID_{RP} and correct endpoint) through a single login flow as IdP only receives PID_{RP} and one-time endpoint, and fails to infer the ID_{RP} from PID_{RP} without the trapdoor t or the RP's endpoint from the independent one-time endpoint.

Moreover, IdP might also try to infer the correlation of RPs in two or more login flows, but fails to classify the accessed RPs for RP's information indirectly. IdP always fails to achieve the relationship between the PID_{RP}s as the secure random number generator ensures the random for generating PID_{RP} and the random string for one-time endpoint are independent in multiple login flows. Therefore, curious IdP fails to classify the RPs based on PID_{RP} and one-time endpoint.

Malicious RP. In the SSO schemes that do not protect user's privacy form collusive RPs, e.g. SPRESSO, collusive RPs are always able to link the same user in multiple RPs through the user identifier (unchanged in different RP) passively. However, these RPs fail to obtain the ID_U directly in UPRESSO.

- These RPs might try to find out the ID_U presenting the unchanged user identity but fails to infer the user's unique information (e.g., ID_U or other similar ones) in the passive way. The PID_U is the only element received by RP that contains the user's unique information. However, RP fails to infer (1) ID_U (the discrete logarithm) from PID_U, due to hardness of solving discrete logarithm; (2) or g^{ID_U} as the r in $ID_{RP} = g^r$ is only known by IdP and never leaked, which prevents the RP from calculating r^{-1} to transfer $Account = ID_{RP}^{ID_U}$ into g^{ID_U}.
- Collusive RPs might try to find out whether the $Account$s in each RP are belong to one user or not but fail to link the user in the passive way. The analysis can only be performed based on $Account$ and PID_U. However, the $Account$ is independent among RPs, as the ID_{RP} chosen by honest IdP is random and unique and the PID_Us are also independent due to the unrelated PID_{RP}

Moreover, the malicious RPs may attempt to link the user actively by tampering with the provided elements (i.e., $Cert_{RP}$, Y_{RP} and PID_{RP}), these RPs still fail to trigger the IdP to generate a same or derivable PID_Us in multiple authentication flows.

- These RPs fail to actively tamper with the messages to make ID_U leaked. IdP fails to lead the PID_U be generated based on the incorrect ID_{RP}, as the modification of $Cert_{RP}$ will make the signature invalid and be found by the user. The malicious RP fails to manipulate the calculation of PID_{RP} by providing an incorrect Y_{RP} as another element n_U is controlled by the user. Also, the malicious RP fails to make an incorrect PID_{RP} (e.g., 1) be used for PID_U, as the honest IdP only accepts a primitive root as the PID_{RP} in the dynamic registration. The RP also fails to change the accepted PID_{RP} in Step 2.3 in Figure **??**, as the user checks it with the cached one.
- Collusive RPs also might lead IdP to generate the PID_Us same or derivable into same $Account$ in each RP. Since the PID_U is generated related with the PID_{RP}, corrupted RPs might choose the related n_{RP} to correlate their PID_{RP}, however, the PID_{RP} is also generated with the participation of n_U, so that RP does not have the ability to control the generation of PID_{RP}. Moreover, corrupted RPs might choose the same ID_{RP} to lead the IdP to generate the PID_U derivable into same $Account$, however, ID_{RP} is verified by the user with through the $Cert_{RP}$, where the tampered ID_{RP} is not acceptable to the honest user.

The user can user the Tor network (or similar means) while accessing the RPs to hide her IP address which prevents collusive RPs to classify the users based on IP addresses, even though currently many network providers only provide user the dynamic IP address based on which the user is unable to be classified.

VII. Implementation and Performance Evaluation

We have implemented the prototype of UPRESSO, and compared its performance with the original OIDC implementation and SPRESSO.

A. Implementation

We adopt SHA-256 to generate the digest, and RSA-2048 for the signature in the $Cert_{RP}$, identity proof and the dynamic registration response. We choose a random 2048-bit strong prime as p, and the smallest primitive root (3 in the prototype) of p as g. The n_U, n_{RP} and ID_U are 256-bit odd numbers, which provides equivalent security strength than RSA-2048 [**?**].

The implementation of IdP only need introduce the minimal modification of existing OIDC implementation. The IdP is implemented based on MITREid Connect [**?**], an open-source OIDC Java implementation certificated by the OpenID Foundation [**?**]. In UPRESSO, we add 3 lines Java code for generation of $PPID$, 25 lines for generation of signature in dynamic registration, modify 1 line for checking the registration token in dynamic registration, while the calculation of ID_{RP}, $Cert_{RP}$, PID_U, and the RSA signature is implemented using the Java built-in cryptographic libraries (e.g., BigInteger)

The user-side processing is implemented as a Chrome extension with about 330 lines JavaScript code and 30 lines Chrome extension configuration files (specifying the required permissions, containing reading chrome tab information, sending the HTTP request, blocking the received HTTP response). The cryptographic calculation in $Cert_{RP}$ verification, PID_{RP} negotiation, dynamic registration, is based on an efficient JavaScript cryptographic library jsrsasign [**?**]. Moreover, the chrome extension needs to construct cross-origin requests to communicate with the RP and IdP, which is forbidden by the same-origin security policy as default. Therefore it is required to add the HTTP header `Access-Control-Allow-Origin` in the response of IdP and RP to accept only the request from the origin `chrome-extension://chrome-id` (`chrome-id` is uniquely assigned by the Google).

We provide the SDK for RP to integrate UPRESSO easily. The SDK provides 2 functions: processing of the user's login request and identity proof parsing. The Java SDK is implemented based on the Spring Boot framework with about 1100 lines JAVA code. The cryptographic computation is completed through Spring Security library. RP processing login request containing identifier negotiation and renewal in Figure **??** and identity proof parsing containing account calculating in Figure **??**.

B. Performance Evaluation

We have compared the processing time of each user login in UPRESSO, with the original OIDC implementation (MITREid Connect) and SPRESSO which only hides the user's accessed RPs from IdP.

We run the evaluation on 3 physical machines connected in a separated 1Gbps network. A DELL OptiPlex 9020 PC (Intel

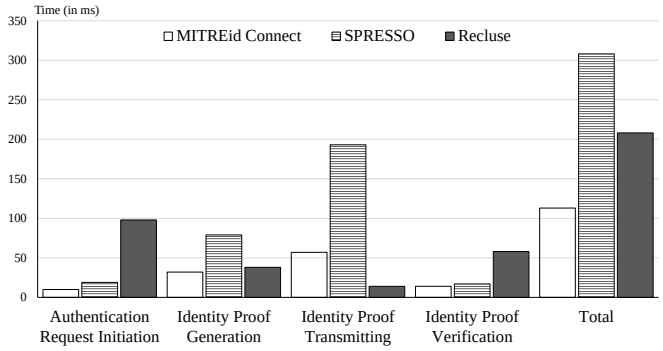

Fig. 5: The Evaluation.

Core i7-4770 CPU, 3.4GHz, 500GB SSD and 8GB RAM) with Window 10 prox64 works as the IdP. A ThinkCentre M9350z-D109 PC (Intel Core i7-4770s CPU, 3.1GHz, 128GB SSD and 8GB RAM) with Window 10 prox64 servers as RP. The user adopts Chrome v75.0.3770.100 as the user agent on the Acer VN7-591G-51SS Laptop (Intel Core i5-4210H CPU, 2.9GHz, 128GB SSD and 8GB RAM) with Windows 10 prox64. For SPRESSO, the extra trusted entity FWD is deployed on the same machine as IdP. The monitor demonstrates that the calculation and network processing of the IdP does not become a bottleneck (the load of CPU and network is in the moderate level).

We have measured the processing time for 1000 login flows, and the the results is demonstrated in Figure **??**. The average time is 208 ms, 113 ms and 308 ms for UPRESSO, MITREid Connect and SPRESSO respectively. The result shows that UPRESSO proved the privacy protection without introducing prominent overhead.

For better comparison, we further divide a SSO login flow into 4 phases, which : 1. **Authentication request initiation** (Steps 1-2.5 in Figure **??**), the period which starts before the user sends the login request and ends after the user receive the identity proof request transmitted from itself. 2. **Identity proof generation** (Step 4 in Figure **??**), denoting the construction of identity proof at the IdP (excluding the user authentication); 3. **Identity proof transmition** (Steps 5.1-5.3 in Figure **??**), for transmitting the proof from the IdP to the RP with the user's help; and 4. **Identity proof verification** (Steps 6 in Figure **??**), for the RP verifying and parsing the proof for the user's $Account$.

In the authentication request initiation, as UPRESSO need negotiate the PID_{RP} (containing 1 modular exponentiation at the user and 2 at the RP) and renew the RP identifier at IdP, it should require the longest time cost. And SPRESSO has to obtain the public information from IdP and encrypt its domain (as the RP identifier), it should result the slight overhead compared with MITREid Connect. Finally, the evaluation shows that MITREid Connect requires the shortest time (10 ms), UPRESSO needs 98 ms and SPRESSO needs 19 ms.

For identity proof generation, UPRESSO need the extra time cost for the generation of PID_U compared with MITREid Connect and SPRESSO. However, the evaluation shows that SPRESSO requires the longest time in this stage, and finally the time cost is found caused by the signature generation as SPRESSO is implemented with JavaScript while the others are using Java. In this stage, MITREid Connect needs 32 ms, UPRESSO needs an extra 6 ms and SPRESSO requires 71 ms.

For identity proof transmission, UPRESSO should need the shortest time among these competitors, as it only need the chrome extension to relay the identity proof from the IdP to RP. MITREid Connect provides the proof as a fragment component (i.e., proof is preceded by #) to RP to avoid the reload of RP document, and RP uses the JavaScript code to send the proof to the background server, which result that the time cost should be at a moderate level. The transmitting in SPRESSO is much complicated: The user's browser creates an iframe of the trusted entity (FWD), downloads the JavaScript from FWD, who obtains the RP's correct URL through a systematic decryption and communicates with the parent opener (also RP's document, but avoiding leaking RP to IdP) and RP's document through 3 post messages. The evaluation result shows that MITREid Connect needs 57 ms, UPRESSO needs only 14 ms and SPRESSO needs about 193 ms.

In identity proof verification, UPRESSO needs the extra time for calculation of $Account$ and signature verification compared with MITREid and SPRESSO. Therefore, the evaluation result is that MITREid Connect needs 14 ms, UPRESSO needs 58 ms and SPRESSO requires 17 ms.

VIII. Discussion

In this section, we provide some discussion about UPRESSO.

Authorization code flow. UPRESSO may be extended to hide the users' access trace in the authorization code flow. The RP obtains the authorization code in the same way as the identity proof in implicit protocol flow. However, the RPs needs to connect to the IdP directly, and use this code with the RP identifier and secret for the `id token`. To avoid the IdP obtaining the IP address from the connection, the anonymous network (e.g., Tor) may be used to establish the connection. And the RP's identifier and secret are issued by the IdP in the dynamic registration described forementioned.

Multi-platform user agent. UPRESSO doesn't store any persistent information in the platform and may be implemented to be platform independent. Firstly, all the information (e.g., $Cert_{RP}$, PID_{RP}, n_U, ID_{RP} and one-time endpoint) processed and cached in the user's platform is only correlated with the current session, which allows the user to log in to any RP with a new platform without any synchronization. Secondly, in the current implementation of UPRESSO, a browser extension is adopted to capture the redirection from the RP and IdP, to reduce the modification at the RP and IdP. However, to comfort the requirement of using UPRESSO in multiple platforms (e.g., mobile phones), UPRESSO is able to be implemented based on HTML5, without the use of any browser extensions,

or plug-ins. The assumption for secure cross-platform user agent is the IdP must always be honest without providing any malicious JavaScript code which is similar with it in SPRESSO (requiring the honest entity, FWD). But it is required the code should be trustful, which is ensured to be correct and unmodifiable by any adversary. As the IdP is considered honest, it could take the responsibility for providing the same trustful JavaScript code as chrome extension to accomplish the PID_{RP} negotiation and other missions. While the code has been already loaded from IdP, if the code is honest (without any prior inserted malicious code) it cannot be modified or monitored. Moreover, the new mechanism called SRI (subresource integrity) under development enables the opener of an iframe to require the hash of document loaded in it to equal with the one set by opener, which ensures the code cannot be malicious even the IdP try to insert the malicious code. For each start, RP opens the iframe with the SRT hash (of correct user agent code) and the iframe downloads the code from IdP, so that, as the RP will never collude with the IdP, the code cannot be malicious.

DoS attack. The adversary may perform the DoS attack. The malicious RPs may try to exhaust the ID_{RP} by applying the $Cert_{RP}$ frequently. However the large p provides a large set of ID_{RP}, and IdP may provide the offline check for $Cert_{RP}$ as it occurs only once for a RP (i.e., the initial registration). The malicious users may attempt to make the other users' PID_{RP} be rejected at the IdP, by registering a large set of PID_{RP}s at IdP. However, the large p makes a huge number of dynamic registration required, and IdP may adopt existing DoS mitigation to limit the number of adversary's dynamic registrations. Moreover, for IdP's dynamic registration storage, the data contains RP's client_id (no more than 256-bit length) and redirect_uri (tens-Byte length). We consider that each dynamic registration data cost no more than 100 Bytes storage. And for each client_id IdP can set the lifetime of validity. It is assumed that for each client_id its lifetime is 5 minutes and during 5 minutes there are 1 billion requests for dynamic registration. So IdP need to offer about 100 GB storage for dynamic registration. The extra cost of storage can be ignored.

Identity injection by malicious IdP. It has been discussed in [**?**] that even the impersonate attack by malicious IdP is not considered, the malicious IdP might lead the user to access the RP as the identity of the adversary (identity injection). That is the IdP might generate an identity proof representing the adversary's identity while an honest user log in to an honest RP. However, in SPRESSO, the user is required to send her email to RP at the very beginning of authentication and IdP must provide the relevant identity proof. It is also available in UPRESSO that user upload her extra user name (defined by user for each RP) before the login to the RP.

RP certificate. The honest IdP is assumed to generate the correct r and ID_{RP}. However, based on the idea of certificate transparency [**?**], an external check may be performed to ensure that no two valid $Cert_{RP}$ assigned to a same ID_{RP} and ID_{RP} is a primitive root modulo p. The external check needs to be performed by a third party instead of RP, as the RP will benefit from incorrect ID_{RP}, e.g., linking the user among RPs with the same ID_{RP}. In UPRESSO, the RP certificate $Cert_{RP}$ is used to provide the trusted binding between the ID_{RP} and the RP's endpoint. RP certificate could be compatible with the X.509 certificate. To integrate RP certificate in X.509 certificate, the CA generates the ID_{RP} for the RP, and combines it in the subject filed (in detail, the common name) of the certificate while the endpoint is already contained. Instead of sending in Step 2.1.2 in Figure **??**, $Cert_{RP}$ is sent to the user during the key agreement in TLS. Moreover, the mechanisms (e.g., the Certificate Transparency) to avoid illegal certificate issued by the CA being adopted to ensure the correctness of ID_{RP}, i.e., gloally unique and being the primitive root.

IX. Related Works

Various SSO standards have been proposed and widely deployed. For example, OIDC is adopted by Google, OAuth 2.0 is deployed in Facebook, SAML is implemented in the Shibboleth project [**?**], and Central Authentication Service (CAS) [**?**] is widely adopted by Java applications. Kerberos [**?**], proposed by MIT, is now replaced by the SSO standards (e.g., OIDC, OAuth) who provide better privacy, as the users in Kerberos fail to control on the releasing of their private information.

A. Security consideration about SSO systems.

Analysis on SSO designing and implementation. Even the user's account at IdP not compromised, various vulnerabilities in the SSO implementations were exploited for the impersonation attack and identity injection, by breaking at least one of the requirements. (1) To break the confidentiality of identity proof, Wang et al. [**?**] performed a traffic analysis/manipulation on SSO implementations provided by Google and Facebook; [**?**], [**?**], [**?**] exploited the vulnerability at the RP's implementations of OAuth, i.e., the publicly accessible information is misused as the identity proof; Armando et al. [**?**] exploited the vulnerability at the user agent, to transmit the identity proof to the malicious RP. (2) The integrity is broken [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] in the implementations of SAML, OAuth and OIDC. For example, [**?**] exploited XML Signature wrapping (XSW) vulnerabilities to modify the identity proof without being found by RPs; the incomplete verification at the client allows the modification of the identity proof [**?**], [**?**], [**?**]; ID spoofing and key confusion make the identity proof issued by the adversary be accepted by the victim RPs [**?**], [**?**]. (3) The designation is also broken [**?**], [**?**], [**?**], [**?**], as the RP may misuse the bearer token as the identity proof [**?**], [**?**], [**?**], and IdP may not bind the refresh/access token with RP which allows the refresh/access token injection [**?**]. Cao et al. [**?**] attempts to improve the confidentiality and integrity, by modifying the architecture of IdP and RP to build a dedicated, authenticated, bidirectional, secure channel between them.

Analysis on mobile platform SSO systems. Compared to web SSO systems, new vulnerabilities were found in the mobile SSO systems, due to the lack of trusted user agent (e.g., the browser) [**?**], [**?**]. The confidentiality of the identity proof may

be broken due to the untrusted transmission. For example, the WebView is adopted to send the identity proof, however, the malicious application who integrates this WebView may steal the identity proof [**?**]; the lack of authentication between mobile applications may also make the identity proof (or index) be leaked to the malicious applications [**?**]. Various automatic tester were proposed to analyze the mobile SSO systems [**?**], [**?**], [**?**], [**?**], [**?**], for the traditional vulnerabilities (e.g., inadequate transmission protection [**?**], token replacement [**?**]) and new ones in mobile platforms (webview [**?**], application logic error [**?**]).

Formal analysis on SSO systems. The comprehensive formal security Analysis were performed on SAML, OAuth and OIDC. Armando et al. [**?**] built the formal model for the Google's implementation of SAML, and found that malicious RP might reuse the identity proof to impersonate the victim user at other RP, i.e., breaking the binding. Fett et al. [**?**], [**?**] conducted the formal analysis of the OAuth 2.0 and OpenID Connect standards using an expressive Dolev-Yao style model, and proposed the 307 redirect attack and IdP Mix-Up attack. The 307 redirect attack makes the browser expose the user's credential to RP. IdP Mix-Up attack allows the malicious IdP to receive the identity proof issued by the correct IdP for the correct RP (who integrates the malicious IdP), which breaks the confidentiality. Fett et al. [**?**], [**?**] proved that OAuth 2.0 and OIDC satisfy the authorization and authentication requirements, as the two bugs are fixed in the revisions of OAuth and OIDC. Ye et al. [**?**] performed a formal analysis on the implementation of Android SSO systems, and found a vulnerability in the existing Facebook Login implementation on Android system, as the session cookie between the user and Facebook may be obtained by the malicious RP application.

Analysis on malicious IdP. One concern of SSO is that, the adversary controls the user's accounts at the correlated RPs, once the user's account at IdP is compromised. A backwards-compatible extension (single sign-off) is proposed for OIDC, which revokes the adversary's access to the RPs [**?**].

The requirements of security authentication are summarized based on the previous work about SSO security. Moreover, as UPRESSO is compatible with OIDC, the protection schemes against existing attacks are also available in UPRESSO.

B. Privacy consideration about SSO systems.

Privacy is the another concern of SSO systems. As suggested in NIST SP800-63C [**?**], the user's privacy protection in SSO systems includes, 1) the user's control on the attributes exposed to the RP, 2) prevention of identity linkage, and 3) avoiding of IdP-based access tracing.

Privacy-preserving SSO systems. OAuth and OIDC provide the user notification to achieve the user's control on its private information [**?**], [**?**]. The pairwise user identifier is proposed to avoid the identity linkage performed by collusive RPs in SAML and OIDC [**?**], [**?**]. In SPRESSO [**?**] and BrowserID [**?**] (adopted in Persona [**?**] and its new version Firefox Accounts [**?**]), IdP doesn't know which RP the user is accessing, however the user's email address is sent to the RP, which introduces the risk of identity linkage performed by the collusive RPs. Fett et al. [**?**], [**?**] performed a formal analysis on the implementation of BrowserID and found that IdP may still know which RP is accessed by the user.

However, none of existing SSO protocols are able to protect user from being tracked by both the collusive RPs and IdP at the same time. Compared with the existing schemes that only protect user's privacy in one side, UPRESSO is able to prevent user from being traces in both sides (being tracked by RPs and IdP). Moreover, UPRESSO is not the simple combining of existing schemes but the completely novel solution based on the OIDC standard.

Anonymous SSO systems. Anonymous SSO scheme is proposed to hide the user's identity to both the IdP and RPs, which may only be applied to the anonymous services that do not identify the user. One of the earliest anonymous SSO system is proposed for Global System for Mobile (GSM) communication in 2008 [**?**]. In 2013, the notion of anonymous single sign-on is formalized [**?**]. Then, the various cryptographic primitives, e.g., group signatures and zero-knowledge proof, are adopted to build anonymous SSO scheme [**?**], [**?**].

However, the anonymous SSO systems enable the user access the service provided by RP without providing her identity to both RP and IdP which avoids user being traced, therefore, RP is unable to distinguish whether multiple accesses are from the same user or not. For most web service providers, it means the personalized service for users are not available, which results the anonymous schemes are not useful. Compared with anonymous SSO schemes, in UPRESSO RP is able to transform the user's PID_U into the constant $Account$, based on which the RP can distinguish the same user in multiple requests.

X. CONCLUSION

In this paper, we, for the first time, propose UPRESSO to protect the users' privacy from both the curious IdP and collusive RPs, without breaking the security of SSO systems. The identity proof is bound with a transformation of the original identifier, hiding the users' accessed RPs from the curious IdP. The user's account is independent for each RP, and unchanged to the destination RP who has the trapdoor, which prevents the collusiveh RPs from linking the users and allows the RP to provide the consecutive and individual services. The trusted user ensures the correct content and transmission of the identity proof with a self-verifying RP certificate. The evaluation demonstrates the efficiency of UPRESSO, about 200 ms for one user's login at a RP in our environment.