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. Our study shows that 80% of the Alexa top-100 websites [**?**] support SSO services and study on the Alexa top 1 million websites has found that 6.30% of websites support SSO [**?**]. 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.

A fundamental requirement of SSO systems is secure authentication [**?**], which should ensure that it is impossible (1) for an adversary to log in to an honest RP as an honest user (i.e., impersonation); and (2) for an honest user to log in to an honest RP as someone else such as an adversary (i.e., identity injection). Extensive study have been performed on existing SSO systems exposing various vulnerabilities [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. It is commonly recognized that SSO security highly depends on secure generation and transmission of identity proofs: (1) the identity proof should be generated in a way that it can never be forged or tampered. For example, when Google used user's attributes with a signature as the identity proof [**?**], the adversary is able to bypass the verification of part of attributes in the identity proof. The root cause is that RP relies on the IdP signed user's attributes to identify the user, however, when the adversary acts as the user, it is able to modify the request and repose between RP and IdP transmitted by user, which allows the adversary add any honest user's attributes in the IdP's response and RP would accept all of them without verifying the signature. Finally the adversary is able to log in to the RP as the honest user. (2) The identity proof should be generated in a way that it is bound to the requesting RP. For example, if an identity proof is generated with nonbinding data such as access tokens in OAuth 2.0, an adversary such as a malicious RP is able to log in to an honest RP as the honest user [**?**], [**?**]. (3) The identity proof should only be obtained by the requesting RP. For example, in some mobile SSO implementations using WebView [**?**], as the url check is not supported which the malicious RP app invokes the IdP's web site in its Webview, the adversary is able to steal the identify proof of an honest user when the app cheats her to consider it as another honest RP.

The wide adoption of SSO also raises new privacy concerns regarding online user tracking and profiling [**?**], [**?**]. In a typical SSO authentication session, for example the OpenID Connect (OIDC) authentication flow as shown in Fig. 1, when a user attempts to log in to an RP, the authentication request is redirected from the RP to the IdP, which generates an identify proof containing information about the user (e.g., user identifier and other authorized user attributes) and the requesting RP (e.g., RP identifier, URL, etc.). If a common user identifier is issued by the IdP for a same user across different RPs or a user's identifiers can be derived from each other, 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 [**?**]. We refer to this as *RP-based identity linkage*. Moreover, when a user leverages the identity issued by one IdP across multiple RPs, the IdP acquires a central role in web authentication, which enables it to collect information about the user's logins

at different sites [**?**]. Since the information of the RP is necessary in the construction of identity proof to make sure it is bound with specific RP [**?**], [**?**], any interested IdP can easily discover all the RPs accessed by a user and reconstruct her access traces by the unique user identifier. We refer to this as *IdP-based access tracing*. Both RP-based identity linkage and IdP-based access tracing could lead to more severe privacy violations.

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 make them easy to link the same user's multiple logins in each RP as the unique user 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 user 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 IdPs from tracking at which RPs users log in, since pseudonymous identifiers are generated by IdP which only hide users' identity from the RPs. To authenticate a user, the IdP has to know her identity. To the best of our knowledge, there are only two approaches (i.e., BrowserID [**?**] and SPRESSO [**?**]) being proposed so far to prevent IdP-based access tracing. Both adopt the idea of hiding RPs' identities from the IdP during the construction and transmission of identity proofs. In particular, the identity proof in BrowserID (and its prototype system known as Mozilla Persona [**?**] and Firefox Accounts [**?**]) is combined with two parts, the binding of the public key generated by the user with her email (as the user identifier) signed by IdP as the user certificate (UC) and the origin of the RP signed by user's private key as the identity assertion (IA). The core idea of BrowserID is that the IdP-generating part of identity proof does not contains any RP's identity and the work of binding identity proof with RP is shift to the user. With the user as the proxy, the IdP does not know RPs' identities throughout the authentication process. In SPRESSO, the RP generates a dynamic pseudonymous identifier by encrypting its domain and a nonce, and passes it to the IdP through the user to create the identity proof, which is returned to the RP through a trusted third party, called forwarder, chosen by the RP itself. While forward transmitting the identity proof, it decrypts the encrypted RP domain to make sure the identity proof only sent to the requesting RP, however, the identity proof is encrypted by another RP generated symmetric key before transmitting to avoid the forward knowing the user's identity from identity proof, which will result in the forward tracing the user. As it designed in SPRESSO, the user email is adopted as the user identifier for identity proof generating.

Unfortunately, none of the existing SSO systems could address both RP-based identity linkage and IdP-based access tracing privacy problems at the same time. The user's identity is represented by several forms in different environments. In detail, the IdP and RP store the unique user identifier (denoted as ID_U in IdP and $Account$ in RP) for each users solely, which are linked by the identity proof containing the junctional user identifier (denoted as ID_U^{Proof}) issued by IdP. Moreover, there is also the unique RP identifier (denoted as ID_{RP}) stored in IdP. In each authentication, the identity proof issued by IdP should contain the ID_{RP} (making sure the identity proof is issued for specific RP) and ID_U^{Proof} (for RP to identify the user). In some traditional SSO systems [**?**], the ID_U and ID_U^{Proof} are the same one, which results in the RP-based identity linkage, and the ID_{RP} is always exposed to IdP, which results in the IdP-based access tracing. However, to avoid the privacy problems in SSO systems, the ID_U^{Proof} should not be the same as the RP (or the ID_U^{Proof} should not be the same in different RPs) and the ID_{RP} should not be exposed to the IdP (using a transformed ID_{RP} representing the RP) in each authentication. The widely adopted SSO standards, such as OIDC and SAML, use $PPID$ as the ID_U^{Proof} transformed from ID_U which is constant in one RP but different in RPs, but expose the ID_{RP} straightforward to the IdP. Otherwise, the SPRESSO adopts the encrypted ID_{RP} (denoted as $enc(ID_{RP})$) and only the $enc(ID_{RP})$ is exposed to the IdP, however, the ID_U^{Proof} is the same one as the ID_U in SPRESSO. As for the BrowserID, the identity proof is generated by the cooperation between IdP and user so that it could avoid exposing ID_{RP} to IdP, but uses same identifier for ID_U and ID_U^{Proof}. The worse thing is there is not a simple way to combine the existing schemes together to protect user's privacy comprehensively (the details are described in Section III).

In this paper, we propose UPRESSO, an Unlinkable Privacy-REspecting Single Sign-On system, as a comprehensive solution to tackle the privacy problems in SSO. We propose novel identifier generation schemes to dynamically generate privacy-preserving ID_U^{Proof} and transformed ID_{RP}, denoted as PID_U and PID_{RP}, to construct identity proofs for SSO. In our scheme, for each login, RP anonymously registers a random ID_{RP} (denoted as PID_{RP}) with IdP, and exposes the PID_{RP} for authentication. IdP generates the PID_U with PID_{RP} and ID_U and issues the identity proof containing PID_{RP} and PID_U. Finally, after RP validates the identity proof, it utilizes the trapdoor obtained when generating the PID_{RP} to derive the $Account$ from PID_U. However the

scheme must satisfy three properties: (1) when a same or differnt user(s) log in to a same RP, random PID_{RP}s are generated in different logins so that a curious IdP cannot infer the real identity of the RP or link multiple logins at that RP; (2) when a same user logs in to a same or different RPs, random PID_Us are generated so that collusive RPs cannot link the logins of that user; (3) when a same user logs in to a same RP, the RP can derive a unique user identifier from different PUIDs with a trapdoor so that it can provide a continuous service to the user during different logins.

Unlike previous approaches that require non-trivial redesign of the existing SSO systems, UPRESSO can be implemented over a widely used OIDC system with small modifications with the support of its dynamic registration function [**?**].

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 level of security 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 II, and the challenges with solutions briefly III. Section IX and Section V describe the threat model and the design of UPRESSO. A systematical analysis is presented in Section VI. We provide the implementation specifics and evaluation in Section VII, then introduce the related works in Section IX, and draw the conclusion finally.

II. Background and Preliminary

A. OpenID Connect

To be compatible with existing SSO systems, UPRESSO is designed on top of OpenID Connect (OIDC) [**?**], one of the most prominent SSO authentication protocols [**?**]. In this section, we introduce OIDC and its implicit flow as an example to describe the authentication flows in SSO. Moreover, our proposed framework can also work for other flows with only micro modification.

OIDC is an extension of OAuth 2.0 to support user authentication. As a typical SSO authentication protocol, OIDC involves three entities, i.e., *users*, *identity provider (IdP)*, and *relying parties (RPs)*. Users register at the IdP to create credentials and identifiers (e.g. ID_U), which are securely maintained by the IdP. Moreover, an RP also has to register at the IdP with its endpoint information to create its unique identifier. During an SSO process, a user is also responsible for redirecting the identity proof request from the RP to the IdP, checking the scope of user attributes in the identity proof returned by the IdP, and forwarding it to the RP. Usually, the redirection and checking are handled by a user-controlled software, called *user agent* (e.g., browser). The IdP maintains user credentials and attributes. Once requested, it authenticates

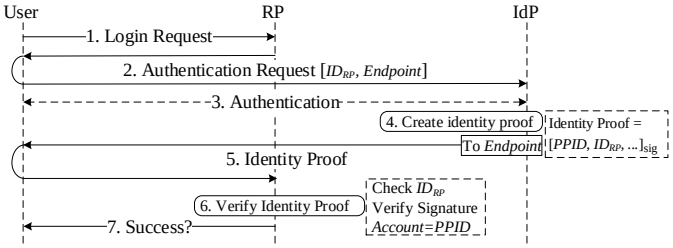

Fig. 1: The implicit protocol flow of OIDC.

the user and generates the identity proof, which contains user identifier (e.g., $PPID$ in OIDC), RP identifier (e.g. ID_{RP}), and a set of user attributes that the user consents to share with the RP. The identity proof is then returned to the registered endpoint of the RP (e.g., URL). RP can be any web server that provides continuous and personalized services to its users. When a user attempts to log in, the RP sends an identity proof request to the IdP through the user, and parses the received identity proof to authenticate and authorize the user.

Implicit flow of user login. OIDC supports three processes for SSO, known as *authorization code flow*, *implicit flow* and *hybrid flow* (i.e., a mix-up of the previous two). In the implicit flow of OIDC, a new token, known as *id token*, is introduced as the identity proof, which contains 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 by its private key to ensure integrity. Moreover, in the authorization code flow, the identity proof is an authorization code bound to the RP. Only the RP with the corresponding secret obtained during the registration at the IdP can extract the user attributes from the identify proof.

As shown in Figure 1, 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 this work we adopt dynamic registration to support the dynamic generated privacy-preserving RP identifier (e.g. PID_{RP}).

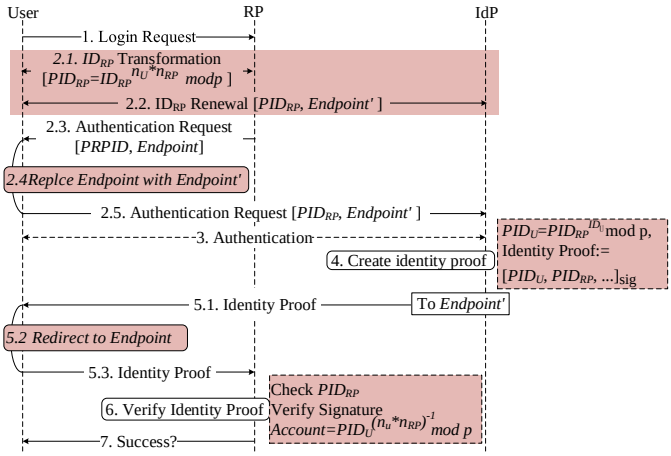

Fig. 2: The UPRESSO.

B. Discrete Logarithm Problem

Discrete logarithm problem is adopted in UPRESSO for privacy-preserving user identifier (e.g. PID_U) and RP identifier (e.g. PID_{RP}) generation. Here, we provide a brief description of the discrete logarithm problem. Given a large prime p, its primitive root g and a number y, it is computationally infeasible to derive the discrete logarithm (here x and $g^x modp = y$) of y (detailed in [**?**]), which is called discrete logarithm problem. The hardness of solving discrete logarithm has been a base of the security of several security primitives, including Diffie-Hellman key exchange and Digital Signature Algorithm (DSA). The PID_U and PID_{RP} generation is based on the modular exponentiation, which requires the parameters, such as ID_{RP} and PID_{RP}, should be the primitive root mod p to prevent the user and RP' identity leakage. To calculate the primitive root for a given large prime p, we retrieve a primitive root g_m mod p, and then calculate the $g = g_m^t mod\ P$, so that g is another primitive root 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

SSO allows users to leverage their existing accounts in the IdP to access services at the RPs. However, building such a system is considered challenging [**?**], as several design and implementation flaws of SSO systems have been discovered, which led to a plethora of attacks [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. In SSO systems, the adversaries' goals described in [**?**] to break the secure authentication are: (1) **Impersonation**: Adversary logs in to the honest RP as an honest user. The adversary might achieve the goal by obtaining a user's identity proof in the ways, such as stealing the proof (from the unprotected HTTP transmission), forging the valid proof (if the integrity is not guaranteed), leading the user to upload a proof valid for other RPs (the proof is not bound with specific RP). (2) **Identity injection**: Honest user logs in to the honest RP under adversaries' identity. The adversary might achieve this goal by replacing the identity transmitted from IdP to RP or lead the user uploads the malicious identity proof in various ways (e.g., CSRF). Based on the formal analysis of SAML, OAuth and OIDC [**?**], [**?**], [**?**] and the study of existing attacks, we summarize basic requirements that we believe a secure SSO platform which means, 1)RP is always able to identify the user; 2)the system should be protected from forementioned attacks, should meet.

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. The anonymous SSO schemes [**?**], [**?**], [**?**] are also proposed to achieve that the user's identity is unknown the neither the IdP nor the RP. However, these schemes are not focally concerned in this paper as the completely anonymous SSO systems are not suitable to the currently web application where the RP need to know the user's identity to provide the personally service. The anonymous SSO schemes are discussed in Section IX.

Receiver designation. The receiver designation requires that the identity proof should be bound to one RP so that it will be accepted only by that RP, and be securely transmitted to the requesting RP (through the user). Otherwise, the adversaries who is able to achieve honest user's identity proof for honest RP, have the ability to conduct impersonation attack [**?**], [**?**]

Integrity. Only the IdP is able to generate a valid identity proof, and no entity should be able to modify or forge it [**?**]. Otherwise, an adversary could modify user identifier in the proof to launch impersonation or identity injection attacks. In OIDC, for example, the IdP signs identity proofs with its private key to provide integrity [**?**], [**?**].

B. The Privacy Dilemma and Existing Attempts

Besides the basic requirements discussed in Section III-A, an SSO system should also provide protection to user privacy against the IdP-based access tracing and RP-based identity linkage. To prevent the former, IdP should not be able to obtain any information identifying the RP accessed by the user (e.g., RP's identifier and URL). To prevent the latter, the collusive RPs should not be able to correlate the identifiers of the user at different RPs in the identity proof.

To conform the requirements of user identification and receiver designation, which is to be validated by the RP, the identity proof must contains (or is related with) specific RP identifier (e.g. ID_{RP}) and user identifier (e.g. ID_U). However, as the identity proof is generated by the IdP, the ID_{RP} and ID_U allows the IdP to know which RP the user accessed. Moreover, as the identity proof is verified by the RP, the colluded RPs are able to link the same user. Therefore, the key point of privacy-preserving SSO system is to hide the ID_{RP} and ID_U while generating the identity proof.

We define that the simplified identity proof model as the tuple $< ID_{RP}, ID_U >$. For each entity in the system,

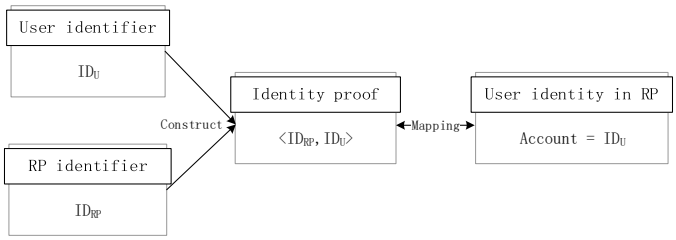

Fig. 3: Traditional SSO.

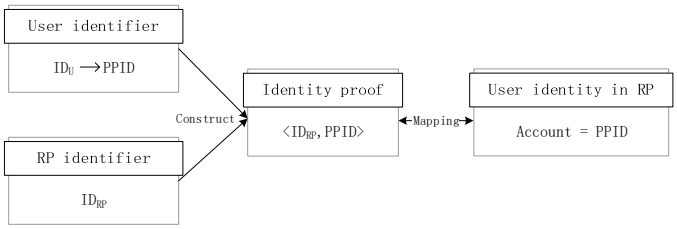

Fig. 4: PPID.

everyone knows the full tuple is able to trace the user (by IdP-based access tracing and RP-based identity linkage). It should be concerned that the RP always know the ID_{RP}. We can transform the problem into hiding the full tuple to IdP and RP.

In traditional SSO systems, the tuple is always exposed to both IdP and RP, so that they do not protect user from any kind of tracing. In OIDC, $PPID$ is adopted to hide the ID_U from RP by using the RP-unique identifier. In SPRESSO, the identity proof is generated for a encrypted ID_{RP} so that the IdP is unable to obtain the full tuple which avoids the IdP-based access tracing. In BrowserID, the generation of identity proof is split and the responsibility of binding it to RP is shifted to user, which also avoids the IdP-based access tracing. However OIDC allows the full tuple is exposed to IdP, while SPRESSO and BrowserID allows the RP obtains the full tuple, all of which cause the user to be tracked.

The key challenge is that there is no simple way to hide the full tuple to both IdP and RP. To tackle this problem, it requires the IdP to generate pairwise identifiers for a user and bind them to pseudo RP identifiers that seem random to IdP, in a way that user identifiers are unique at one RP but different at different RPs and user identifiers bound to pseudo identifiers of the same RP can be associated by that RP.

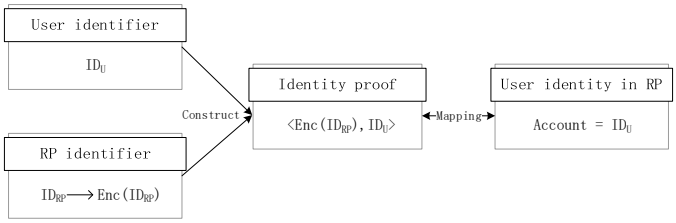

Fig. 5: SPRESSO.

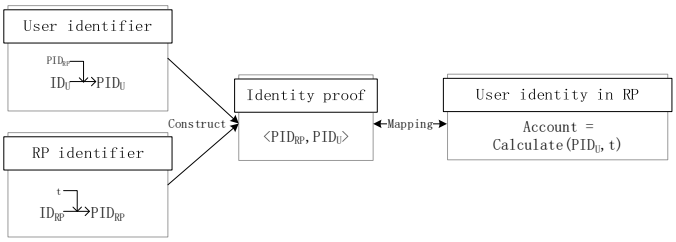

Fig. 6: Our scheme.

C. The Principles of UPRESSO

In this work, we present UPRESSO to defend against both IdP-based access tracing and RP-based identity linkage privacy attacks with enhanced security for identification, binding and confidentiality. Since the integrity requirement for identity proof is orthogonal to the privacy requirement, UPRESSO adopts the same signature-based integrity mechanism of OIDC. Next, we overview the new privacy-preserving user identification and receiver designation schemes in UPRESSO.

Trapdoor user identification. The trapdoor identification make the RP possible to transform the user's identity proof into the constant Account in RP, even the ID_U^{Proof} (user identifier in identity proof) is changing with the PID_{RP}, which is to prevent the RP-based identity linkage.

Transformed receiver designation. The pseudonymous receiver uniqueness allows the IdP to generate the identity proof bound to specific RP without knowing any straightforward RP information but a transformed RP identifier, and the identity proof is bound to the transformed identifier. As the IdP does not the RP's identity, responsibility of url checking should be shifted from IdP to RP, which guarantees the identity proof is only sent to the bound RP.

To combine these three principles into one system, we propose novel identifier generation schemes to dynamically generate transformed RP identifier and trapdoor identity proof, which makes the RP's information never exposed to the IdP during the authentication and RP able to derive the unique user $Account$ from identity proof. Moreover, both the RP identifier transforming and identity proof transformation (including url check) require the trustful user agent. We do not explain how to finish each principle here, it will be explained in Section V-A.

IV. Threat Model and Assumption

There are three types of entities in UPRESSO, i.e., the IdP, a group of RPs and users. Unlike existing privacy-preserving SSOs [**?**], UPRESSO does not rely on any trusted third parties.

A. Threat Model

While the IdP is assumed to be semi-honest, the users and RPs can be malicious or even collude with each other. However, even with the presence of malicious RPs and users in the login flows, UPRESSO is expected to protect honest users from (1) being impersonated by the malicious user to log in to the honest RPs, and (2) logging in to the honest RPs

as the malicious user. In particular, the adversary attempts to break the security and privacy properties under the following threat model.

Honest but curious IdP. We assume the IdP is well-protected and its private key for signing the RP certificate and identity proof is never leaked. The IdP always processes messages correctly and only conducts the necessary actions defined by the protocol. Moreover, the IdP never colludes with malicious RP nor user. However, the IdP is also curious about users' private information. It may attempt to infer the user's access traces (i.e., which RPs accessed by the user) by analyzing the content and timing of the received messages. For example, in OIDC IdP achieves the user's identity while authenticating the user and knows user accessed RP by the RP's identifier and endpoint in the authentication request.

Malicious user. The adversary may obtain the user's credential through various attacks, or register a valid account at the IdP and the RPs. The user controlled by the adversary may behave arbitrarily, attempting to perform the impersonation attacks and identity injection. For example, the malicious user may attempt to lead the RP to accept an identity proof issued for another RP by sending illegal login request to the RP, transmitting a modified or forged identity proof request to the IdP [**?**], or choosing a non-random nonce to participate in the generation of RP's transformation identifier. Moreover, the malicious user also attempts to create the valid identity proof for an honest RP itself by replying a corrupted or forged identity proof to the RP [**?**], [**?**].

Malicious RP. The adversary may work as a RP itself, by controlling one or multiple compromised RPs, or registering as valid RPs with the IdP. The malicious RPs may behave arbitrarily to break the principles of the identity proof. For example, the malicious RP may attempt to achieve the (current authenticated) user's identity proof valid for other honest RPs or lead the IdP to issue the non-privacy-preserving identity proof (resulting in RP-based identity linkage), by deliberately choosing certain nonces in RP identifier transformations and trick honest RPs to accept them, provide an incorrect identity proof request with incorrect RP identifiers, or send the invalid or other RP's valid certificate instead of its own.

Collusive user and RP. In particular, the malicious users and RPs may collude to perform the impersonation or identity injection attacks. For example, the adversary may first attract the victim user to access a malicious RP to initiate an authentication request for identity proof that is also valid to honest RPs, and then pretend to be the victim user to access these RPs using the received identity proof. The adversary may also act as a user to obtain an identity proof for herself to access an honest RP, and then work as an RP to redirect the victim user to the honest RP using her proof (e.g. CSRF).

B. Assumption

To break user's privacy, the collusive RPs may link user accounts across them by actively returning incorrect messages, or passively combining the received messages. However, user linkage based on other attributes in user accounts and the global network traffic analysis are not considered in this work, which may be prevented by limiting the attributes exposed to each RP and introducing cover traffic by accessing irrelevant RPs.

We assume the user agent deployed at the honest user is correct, and will transmit the messages to the correct destination without leakage. The TLS is 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 the random numbers in UPRESSO are unpredictable by using a secure random number generator; and the adopted cryptographic algorithms, including the RSA and SHA-256 required in UPRESSO, are secure and implemented correctly, that is, no one without private key can forge the signature, and the adversary fails to infer the private key. Moreover, also the Discrete Logarithm Problem in Section II-B.

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 III-C, 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 I.

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 VI. 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 2). 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).

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 3. 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 4 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**.

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

TABLE I: The notations used in UPRESSO.

Notation	Definition	
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

C. Overall protocol flow overview

In this section, we present the detailed process for each user's login as shown in Figure 8.

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 **??**. **(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 V-D) 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 V-D). **(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 VIII).

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

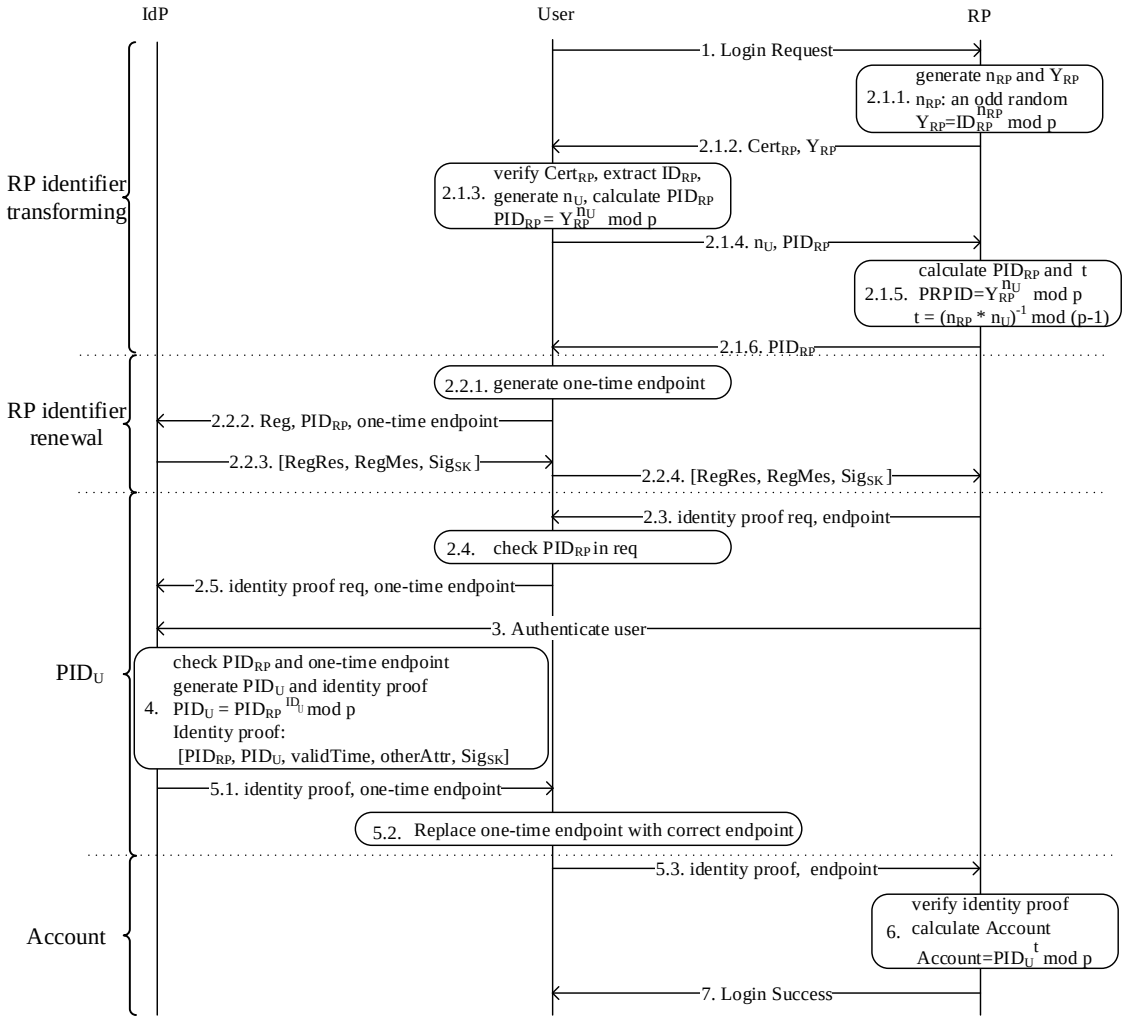

Fig. 7: Process for each user login.

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 II) 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 8, 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 8 and identity proof parsing containing account calculating in Figure 8.

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 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 9. 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 8), 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 8), denoting the construction of identity proof at the IdP (excluding the user authentication);

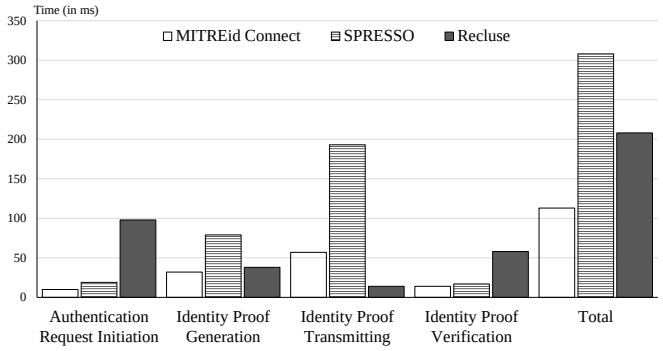

Fig. 8: The Evaluation.

3. **Identity proof transmition** (Steps 5.1-5.3 in Figure 8), for transmitting the proof from the IdP to the RP with the user's help; and 4. **Identity proof verification** (Steps 6 in Figure 8), 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 8, $Cert_{RP}$ 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.