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 colluding 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 colluding 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 colluding 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 [11], OpenID Connect [7] and SAML [44], 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. According to Alexa [1], 80% of the top-100 websites support SSO services. 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 [6], 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[cite]. 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 [cite], the adversary is able to bypass the verification of part of attributes in the identity proof. (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 [cite], an adversary is able to steal the identify proof of an honest user when she requests to log in to an honest RP using the malicious app.

Besides, the wide adoption of SSO also raises new privacy concerns regarding online user tracking and profiling [?], [2]. 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 some widely deployed SSO systems [6], [9], colluding 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, 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 [3], Onavo [4]). By simply serving the IdP role, these companies can easily collect a large amount of continuous data to reconstruct users' online traces. Moreover, many IdPs are also service/content providers hosting a variety of web services. 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 [**?**], [2], only a few solutions have been proposed to protect user privacy [5], [6]. Among them, Pairwise Pseudonymous Identifier (PPID) [7], [8] 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 [2], PPID has been specified in many widely adopted SSO standards including OIDC [7] and SAML [8].

However, PPID-based approaches cannot prevent the IdPs from tracking at which RPs users log in, since pseudonymous identifiers 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 [9] and SPRESSO [6]) 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, BrowserID (and its prototype system known as Mozilla Persona [5] and Firefox Accounts [45]) uses email address as user identifier, and lets the IdP bind the public key generated by the user with her email. The identity proof is signed with the corresponding private key of the user and transmitted to the RP via 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.

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. In BrowserID and SPRESSO, colluding RPs could link a user's multiple logins from the common user identifier. In this paper, we present 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 user and RP identifiers, denoted as $PUID$ and $PRPID$, to construct identity proofs for SSO, which satisfy three properties: (1) when a same or diffent user(s) log in to a same RP, random $PRPID$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 $PUID$s are generated so that colluding 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 [10].

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 colluding 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

To be compatible with existing SSO systems, UPRESSO is designed on top of OpenID Connect (OIDC) [7], one of the most prominent SSO authentication protocols [8]. In this section, we introduce OIDC and its implicit flow as an example to describe the authentication flows in SSO, and overview the basic security requirements for general SSO systems.

A. OpenID Connect (OIDC)

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, which are securely maintained by the IdP. 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 the user and generates the identity proof, which contains user identifier (e.g., PPID in OIDC), RP identifier, 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. In SSO, an RP also has to register at the IdP with its endpoint information. 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.

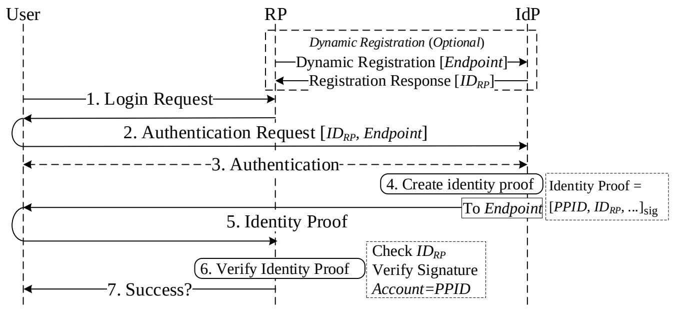
User
RP
IdP
Dynamic Registration (Optional)
Dynamic Registration [Endpoint]
Registration Response [ID_{RP}]
1. Login Request
2. Authentication Request [ID_{RP}, Endpoint]
3. Authentication
4. Create identity proof
Identity Proof = [PPID, ID_{RP}, ...]$_{sig}$
To Endpoint
5. Identity Proof
Check ID_{RP}
Verify Signature
Account=PPID
6. Verify Identity Proof
7. Success?

Fig. 1: The implicit protocol flow of OIDC.

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).

Single Sign-on. 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., RPID), 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 RPID, 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).

Dynamic Registration. OIDC provides a dynamic registration mechanism [10] for the RP to renew its RPID 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 RPID from the IdP.

B. Basic Requirements of a Secure SSO System

SSO allows users to leverage their existing accounts in the IdP to access services at the RPs. However, in SSO systems, the adversaries' goals described in [6] 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). Therefore, building such a system is considered challenging [6], as several design and implementation flaws of SSO systems have been discovered, which led to a plethora of attacks [14], [18]–[29]. Based on the formal analysis of SAML, OAuth and OIDC [15]–[17] and the study of existing attacks, we summarize basic requirements that we believe a secure SSO platform should meet.

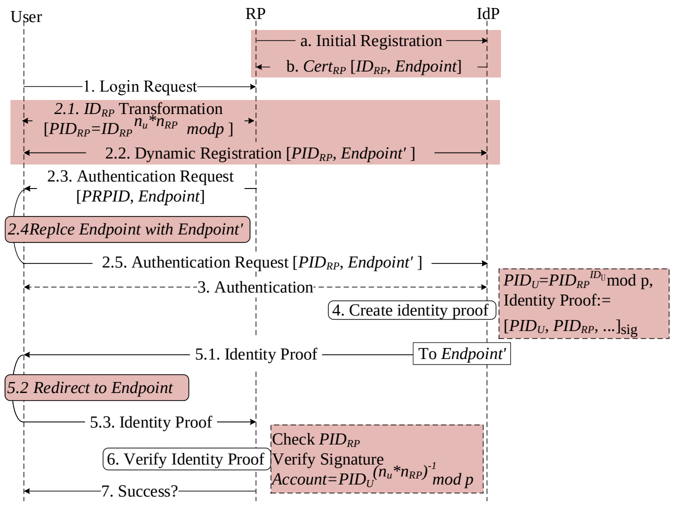
User
RP
IdP
a. Initial Registration
b. $Cert_{RP}$ [ID_{RP}, Endpoint]
1. Login Request
2.1. ID_{RP} Transformation
[$PID_{RP}=ID_{RP}^{n_u * n_{RP}}$ modp]
2.2. Dynamic Registration [PID_{RP}, Endpoint']
2.3. Authentication Request [PRPID, Endpoint]
2.4Replce Endpoint with Endpoint'
2.5. Authentication Request [PID_{RP}, Endpoint']
3. Authentication
$PID_U=PID_{RP}^{ID_U}$mod p,
Identity Proof:=
[PID_U, PID_{RP}, ...]$_{sig}$
4. Create identity proof
5.1. Identity Proof
To Endpoint'
5.2 Redirect to Endpoint
5.3. Identity Proof
Check PID_{RP}
Verify Signature
Account=$PID_U^{(n_u * n_{RP})^{-1}}$ mod p
6. Verify Identity Proof
7. Success?

Fig. 2: The UPRESSO.

Identification. After a successful SSO authentication, the RP should be able to uniquely identify the user. 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. This is an essential requirement for any authentication mechanism. In SSO, this indicates that a same unique user identifier should be derived from multiple identity proofs of the user by the RP. In OIDC, for example, a user obtains different PPIDs from the IdP, each for a different RP.

Additionally, the anonymous SSO schemes [47]–[49] 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.

Binding. Each identity proof should be bound to one RP so that it will be accepted only by that RP. Otherwise, the identity proof may be abused to impersonate the victim user at other honest RPs [13], [14]. To achieve such binding in OIDC, the IdP embeds the identifier of the target RP in the identity proof and signs it with its private key. When receiving an identity proof, the RP needs to check if it is the intended receiver.

Confidentiality. The identity proof should be securely returned to the requesting RP (and the user). Otherwise, an adversary who obtains an identity proof can impersonate the user at the RP specified in the identity proof [13], [14], [16]. Currently, OIDC adopts TLS to ensure the confidentiality during the transmission. However, additional checking should be performed during the generation of the identity proof by the IdP and the transmission of the identity proof by the user and its trusted user agent.

Integrity. Only the IdP is able to generate a valid identity proof, and no entity should be able to modify or forge it [14]. 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 [18], [19].

III. Problems in Privacy-Preserving SSO

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

A. The Problem

Besides the basic requirements discussed in Section II-B, 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 colluding RPs should not be able to correlate the identifiers of the user at different RPs in the identity proof.

The root reason that existing SSO systems suffer from privacy attacks is that they cannot meet secure authentication and privacy protection requirements at the same time. For secure authentication, the identity proof should be bound to a specific RP identifier and a specific user identifier. However, for privacy protection, the privacy-preserving SSO systems require (i) the RP identifier to be one-time and indistinguishable to IdP, and (ii) the user identifier to be globally indistinguishable to IdP but identifiable to the RP.

Several privacy-preserving schemes have been proposed to meet the privacy requirements while supporting secure authentication, however, they cannot satisfy both privacy requirements at the same time. For example, OIDC adopts PPID as an RP-specific user identifier, but for secure authentication, it has to use a unique RPID to generate the identity proof and thus suffers from IdP-based tracing. On the contrary, SPRESSO encrypts the RP domain to generate a one-time RPID that hides the RP identity against IdP, but it uses static user identifier in identity proof and thus is vulnerable to RP-based linkage. BrowserID takes a different approach by binding the globally unique user identifier with an asymmetric key pair and letting the user to sign RP's domain with the private key. However, the unique signature can be used to link the RPs accessed by the same user in RP-based linkage.

The key challenge is that there is no simple way to bind the identity proof to a user's identity without exposing the requesting RP's identity to IdP. 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.

B. UPRESSO Overview

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 identification, binding and confidentiality schemes in UPRESSO.

Trapdoor Identification. The entity with the trapdoor can derive the unique user account associated with an RP from different identifiers in multiple identity proofs generated when the user logs in to that RP multiple times. In UPRESSO, the user's account at an RP is a function of the RP's original identifier and the user's unique identifier. The generation of this unique user account consists of two steps executed by the IdP and RP independently to prevent not only the IdP from obtaining the RP identifier but also the RP from inferring the unique user identifier. In the first step, IdP generates a PPID with the user identifier and the RP identifier transformation. Since the RP identifier transformations are independent in multiple login flows, the PPIDs generated by the IdP in these flows are different. In the second step, the requesting RP adopts the trapdoor of the transformation of the RP identifier to derive the unique user account from the PPIDs. Moreover, the accounts of a user at various RPs are different, as the original RP identifiers are different, which prevents the identity linkage.

Transformed Binding. The identity proof is bound to the transformation of a RP identifier, which allows the RP to check whether the identity proof is for it while preventing the IdP from inferring the original RP identifier. Only the identity proof bound to a fresh transformation of its identifier will be accepted by the RP. In UPRESSO, the transformation of the RP identifier is generated by the user and RP cooperatively, which prevents the adversary from constructing a same or related transformation for different RPs. The IdP obtains the transformation of an RP identifier and binds it to the identity proof.

User-centric Confidentiality. In UPRESSO, the user sends the authentication request to the IdP to generate an identity proof for the requested RP identifier transformation. To ensure the identity proof is securely returned to the requesting RP, UPRESSO adopts a user-centric confidentiality check, which requires the user to extract RP's identifying information (e.g., URL, name and RP identifier) from the RP certificate issued by a trusted entity (e.g., IdP). Therefore, it is impossible for an adversary to request the identity proof on behalf of others or intercept others' identify proofs with a forged return URL.

IV. Threat Model and Assumption

There are three types of entities in UPRESSO, i.e., the IdP, a group of RPs and users. Unlike other 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 never colludes with other malicious entities. More specifically, for RP's initial registration, the IdP checks the correctness of RP's URL, assigns an unique original identifier, and generates the correct signature. For identity proof, the IdP generates the proof only for the authenticated user, calculates the user identifier based on the user's unique identifier and the user-provided transformation of RP identifier, binds the proof with the transformation, generates the signature correctly, and sends it only to the user. 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.

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 send illegal login request to the RP, transmit a modified or forged identity proof request to the IdP, reply a corrupted or forged identity proof to the RP, or choose a non-random nonce to participate in the generation of RP's transformation identifier.

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 integrity, confidentiality and binding of the identity proof. For example, they may manipulate the generation of identity proofs 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.

Colluded RPs and users. 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).

Colluded RPs. To break user's privacy, the colluding 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.

B. Other Assumptions

In UPRESSO, 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, according to the Discrete Logarithm Problem, we assume the adversary fails to infer r from $g^r mod\ p$, where p is a large prime and g is the primitive root, which is required by the the transformation of the RP's identifier and trapdoor identification.

V. Design of UPRESSO

In this section, we first present the design goals of UPRESSO with the main process phases. Then, we describe system initialization and initial registration of RP, followed by 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. Design Goals

In Section III-B, we highlight three design features of UPRESSO, namely trapdoor identification, transformed binding and user-centric confidentiality, to develop a secure and privacy-preserving SSO. Next, we will discuss three design goals that directly support these features, and explain how each goal is achieved correspondingly along the authentication flow of UPRESSO, as shown in Figure 2. To ease the presentation, we list the notations used in this paper in Table I.

Goal 1: *Generating an unforgeable and self-verifiable information for each RP to provide correct information for user-centric checking.*

In UPRESSO, such information is contained in the RP certificate ($Cert_{RP}$), which is issued by the IdP in the *RP initial registration* phase (Steps a and b in Figure 2). In each login process, the user compares the information about the RP extracted from the RP certificate and the identity proof and verifies that the identity proof issued by the IdP is generated for the requesting RP. This provides **user-centric confidentiality**.

Goal 2: *Generating a unique, session-specific and privacy-preserving RP identifier for the IdP to bind the identity proof to it.*

In UPRESSO, 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). As a result, non-colluding adversaries cannot view or forge this identifier, nor tamper or control its generation. Meanwhile, UPRESSO requires IdP to check the uniqueness of every PID_{RP} it receives in the *dynamic registration* phase (Step 2.2 in Figure 2), so that each identity proof is bound to one unique PID_{RP}. In this way, UPRESSO supports **transformed binding**.

Goal 3: *Generating a 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 from multiple privacy-preserving user identifiers for the same user at different RPs.*

In UPRESSO, IdP computes a unique privacy-preserving user identifier (denoted as PID_U) based on the original user identifier (i.e., ID_U) and the privacy-preserving RP identifier (i.e., PID_{RP}) in the PID_U *generation* phase (Step 4 in Figure 2). From each PID_U, an RP can derive a value (denoted as $Account$) with the trapdoor, i.e., the secret parameters of the RP identifier transformation, in the $Account$ *processing* phase (Step 6 in Figure 2). $Account$ uniquely identifies the user at this RP, so that it can be used to link all PID_U converted from the same ID_U. It is worth noting that $Account$ is different from ID_U and it provides no clue to derive ID_U, therefore, the RP still does not know the real user associated with the identity proof. Moreover, to colluding RPs, the PID_Us for a same user are seemingly irrelevant to each other, since they are associated with different $Account$s at these RPs. Therefore, UPRESSO supports **trapdoor identification**.

Discrete logarithm problem is adopted in UPRESSO, to prevent IdP from inferring ID_{RP} from PID_{RP}, and allow the RP to derive $Account$ from PID_U without obtaining the ID_U. 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 $(0 \leq x < p-1)$ satisfying $y = g^x \pmod p$. And, x is called the discrete logarithm of y modulo p. Given a large prime p and a number y, it is computationally infeasible to derive the discrete logarithm (here x) of y (detailed in [30]), 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). To calculate the primitive root for a given large prime p, we retrieve the least primitive root g_m mod p, and then calculate the primitive root $g = g_m^t mod\ P$, where t is an integer coprime to $p-1$. A lemma is proposed to check whether an integrity μ is the primitive root modulo p where $p = 2q+1$ (q is a prime), that is, an integer $\mu \in (1, p-1)$ is a primitive root if and only if $\mu^2 \neq 1\ mod\ p$ and $\mu^q \neq 1\ mod\ p$ [31], [32].

In UPRESSO, a system initialization is performed to initialize the IdP, which generates g, p, PK_{Cert}, SK_{Cert}, PK_{ID} and SK_{ID}, and provides g, p, PK_{Cert} and PK_{ID} as the public parameters. Only one initial registration is performed by each RP to apply $Cert_{RP}$ and ID_{RP} from IdP. The user triggers Steps 1-7 in Figure 2 for each login at a RP.

In UPRESSO, a system initialization is performed to initialize the IdP only once at the very beginning of system construction and generates the strong prime p, the primitive root g, and the key pairs (PK_{Cert}, SK_{Cert}, PK_{ID} and SK_{ID}) for RP certificate and identity proof generation. The initial registration is performed by RP to apply $Cert_{RP}$ (for user to verify the attributes of RP) and ID_{RP} (for the transformation of ID_{RP}) from IdP only when the new RP is built. Besides, only the user login process are conducted in each login flow, containing the transformation (PID_{RP}) negotiation, the dynamic registration and the modified OIDC implicit flow.

TABLE I: The notations used in UPRESSO.

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

B. System Initialization and RP Initial Registration

System initialization. The IdP generates two random asymmetric key pairs, (PK_{ID}, SK_{ID}) and (PK_{Cert}, SK_{Cert}), for calculating the signatures in the identity proof and $Cert_{RP}$, respectively; and provides PK_{ID} and PK_{Cert} 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 V-C). Once SK_{Cert} or SK_{ID} leaked, IdP may update these two asymmetric key pairs. 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 ($Sig_{SK_{Cert}}$) of $[ID_{RP}, Name_{RP}]$ with SK_{Cert}, and returns $[ID_{RP}, Name_{RP}, Sig_{SK_{Cert}}]$ as $Cert_{RP}$.
3) IdP sends $Cert_{RP}$ to the RP who verifies $Cert_{RP}$ using PK_{Cert}.

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.

C. RP identifier transformation and Account calculation

In this section, 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 [33], 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**.

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

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 colluded 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:

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

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 $PID_{RP}^{ID_U}$.

$$PID_U = {PID_{RP}}^{ID_U}\ mod\ p \qquad (3)$$

Account. 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**.

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

D. User Login Process

In this section, we present the detailed process for each user's login as shown in Figure 3.

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)** RP chooses the random n_{RP}, and calculates Y_{RP} as described in Section V-C. **(3)** RP sends $Cert_{RP}$ with Y_{RP} to the user. **(4)** 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 V-C. **(5)** The user sends n_U and PID_{RP} to the RP. **(6)** RP calculates PID_{RP} using the received n_U with Y_{RP} as in Section V-C, and rejects the user's login request if the calculated PID_{RP} is inconsistent with the received one. After that, RP derives the trapdoor t as in Section V-C, which will be used in calculating $Account$. **(7)** 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 dynamic registration phase, the user registers the newly generated PID_{RP} at the IdP as follows. **(8)** The user generates an one-time endpoint (used in Section V-E) if the received PID_{RP} is accepted. **(9)** Then, the user registers the RP with the PID_{RP} and one-time endpoint. **(10)** 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 $[RegRes, RegMes, Sig_{SK_{ID}}]$ where $RegMes$ is the response to traditional dynamic registration containing PID_{RP}, issuing time as well as other attributes

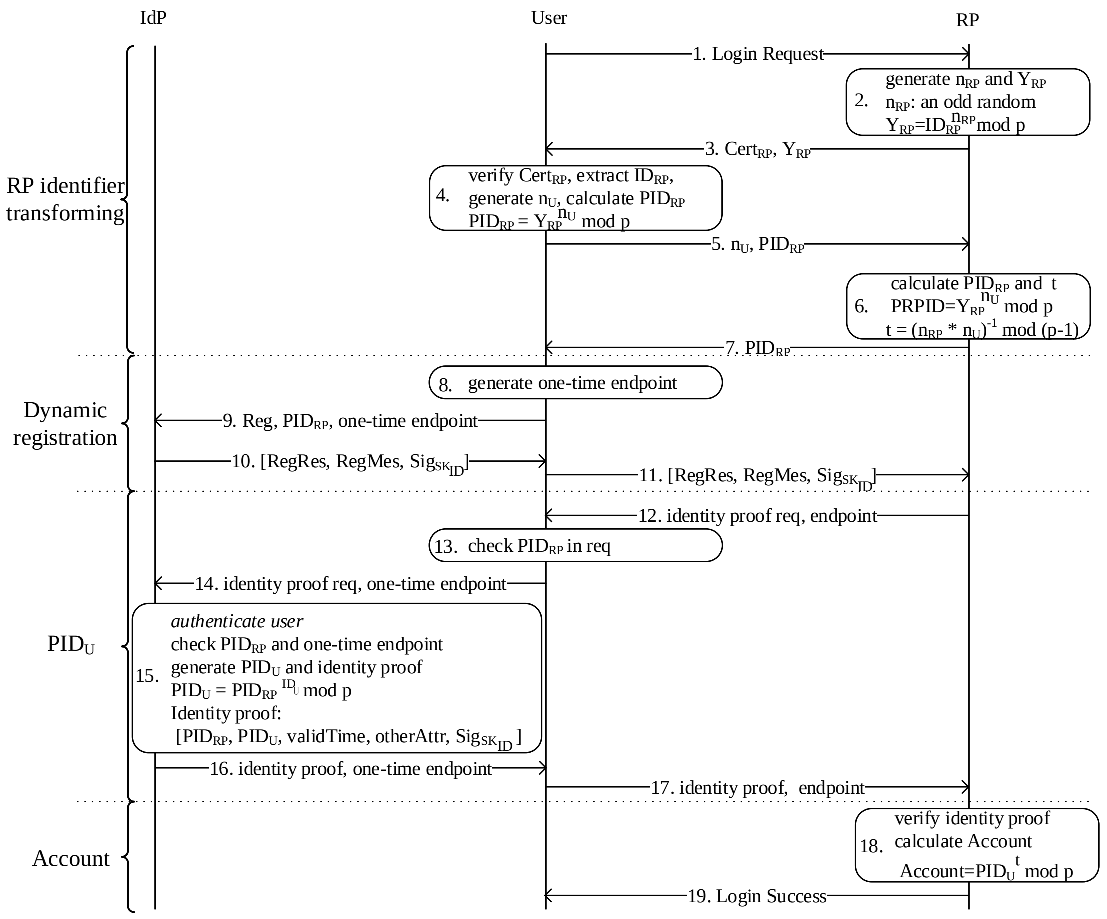
IdP
User
RP
1. Login Request
2. generate n_RP and Y_RP
n_RP: an odd random
Y_RP=ID_RP^n_RP mod p
3. Cert_RP, Y_RP
4. verify Cert_RP, extract ID_RP,
generate n_U, calculate PID_RP
PID_RP = Y_RP^n_U mod p
5. n_U, PID_RP
6. calculate PID_RP and t
PRPID=Y_RP^n_U mod p
t = (n_RP * n_U)^-1 mod (p-1)
7. PID_RP
RP identifier transforming
8. generate one-time endpoint
9. Reg, PID_RP, one-time endpoint
10. [RegRes, RegMes, Sig_SK_ID]
11. [RegRes, RegMes, Sig_SK_ID]
Dynamic registration
12. identity proof req, endpoint
13. check PID_RP in req
14. identity proof req, one-time endpoint
15. authenticate user
check PID_RP and one-time endpoint
generate PID_U and identity proof
PID_U = PID_RP^ID_U mod p
Identity proof:
[PID_RP, PID_U, validTime, otherAttr, Sig_SK_ID]
PID_U
16. identity proof, one-time endpoint
17. identity proof, endpoint
18. verify identity proof
calculate Account
Account=PID_U^t mod p
19. Login Success
Account

Fig. 3: Process for each user's login.

and $Sig_{SK_{ID}}$ is the signature of the other elements using the private key SK_{ID} (ensuring unique PID_{RP} for binding in **Goal 2**). **(11)** 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 $Sig_{SK_{ID}}$ 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. **(12)** 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-E). **(13)** The user halts the login process if the received PID_{RP} is different from the previous one. **(14)** The user replaces the endpoint with the registered one-time endpoint, and sends it with the identity proof request to the IdP. **(15)** IdP requires the user to provide the correct credentials if the user hasn't been unauthenticated; and 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 V-C, 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_{ID}. **(16)** IdP sends the identity proof with the one-time endpoint to the user. **(17)** 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. **(18)** RP accepts the identity proof only when the signature is correctly verified with PK_{ID}, 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 V-C. **(19)** RP sends the login result to the user and begins to provide the personalized service.

E. Compatibility with OIDC

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

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_{ID}.

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).

The RP's extra processes needed in UPRESSO may be achieved using the existing interfaces defined in OIDC. UPRESSO requires that PID_{RP} is globally unique, which can be achieved through the dynamic registration (described in Section II) provided by IdP in OIDC. In UPRESSO, the dynamic registration is invoked by the user instead of the RP to prevent the curious IdP linking the dynamic registration from RP with the identity proof request sent by the user. To avoid the IdP to infer the RP's identity through the registration token in dynamic registration, a common registration token is used in UPRESSO. 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. Moreover, the endpoint in the dynamic registration request is replaced with an one-time endpoint, to avoid the RP's identifying information to be leaked to the IdP.

The modification of the RP required by UPRESSO may be achieved only using the existing interfaces provided by the current implementations of OIDC. Based on the software development kit (SDK) in existing OIDC implementations for RP, the Steps 2-3, 5-7, 12 (in Figure 3) in RP may be integrated in the interface for constructing identity proof request; Step 18 in Figure 3 may be combined with the interface for parsing identity proof.

The processes at the user may be achieved through an extension at the user agent, which captures the identity proof (and request) without modifying existing message transmission at the IdP and RP (i.e., redirection mechanism).

Only the RP initial registration, step 10 and Step 15 in Figure 3 require the modification at IdP.

In the actually deployed systems, the comprehensive update of RPs would be the long term procedure, during which there might be the updated and legacy RPs at the same time. In USPRESSO, the legacy OIDC flow started by the legacy user agent (starting the login by clicking the button in the RP's web page) is supported as it only requires the step 1, 12, 14, 15, 16 and 17 in Figure 3 and the process is accepted by the IdP and legacy user agent, only requiring the IdP to tag the legacy ID_{RP} and provide the legacy PPID to these ID_{RP}.

VI. Analysis

In this section, we firstly prove the privacy of UPRESSO, i.e., avoiding the identity linkage at the colluded 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 [17].

A. Privacy

Curious IdP. The curious IdP might be interested in the user accessed RP or infer the correlation of RPs in two or more login flows by performing the analysis on the content and timing of received messages. However, it fails to obtain the user's accessed RPs directly, nor classifies the accessed RPs for RP's information indirectly.

- The curious IdP might be interested in the RP's identity but fails to derive RP's identifying information (i.e., ID_{RP} and correct endpoint) through a single login flow. 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.
- It also might try to infer the correlation of RPs in two or more login flows but fails achieve the relationship between the PID_{RP}s. 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.
- It even fails to obtain the correlation of RPs through analyzing the timing of received messages. IdP fails to map user's accessed RP in the identity proof to the origin of dynamic registration based on timing, as both the dynamic registration and the identity proof request are sent by the user instead of the RP.

Malicious RPs. The malicious RPs may attempt to link the user passively by combining the PID_Us received by the colluded RPs, or actively by tampering with the provided elements (i.e., $Cert_{RP}$, Y_{RP} and PID_{RP}). However, these RPs still fail to obtain the ID_U directly, or trigger the IdP to generate a same or derivable PID_Us.

- A single RP 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}.
- A single RP fails to actively tamper with the messages to make ID_U leaked. The modification of $Cert_{RP}$ will make the signature invalid and be found by the user. The malicious RP fails to control 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 11 in Figure 3, as the user checks it with the cached one.

- Two or more 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}.
- Two or more 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.

Colluded RPs even fail to correlate the users based on the timing of users' requests, when the provided services are unrelated. For the related services, (e.g., the online payment accessed right after an order generated on the online shopping), the user may break this linking by adding an unpredicted time delay between the two accesses. The anonymous network may be adopted to prevent colluded RPs to classify the users based on IP addresses.

B. Security

UPRESSO protects the user's privacy without breaking the security. That is, UPRESSO still prevents the malicious RPs and users from breaking the identification, integrity, confidentiality and binding of identity proof.

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 confidentiality of identity proof, UPRESSO inherits the same idea from OIDC, i.e., TLS, a trusted user agent and the checks. 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 with the registered ones, preventing the adversary from triggering the IdP to generate an incorrect proof or transmit 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 that the ID_{RP} in identity proof is a fresh PID_{RP} negotiated based on the ID_{RP} and the endpoint is the one-time one corresponding to 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 ID_{RP} and PID_U. The correct RP checks the binding by comparing the PID_{RP} with the cached one, and provides the service to the $Account$ based on PID_U.

UPRESSO binds the identity proof with PID_{RP}, instead of a random string unique for each RP assigned by IdP in OIDC. However, the adversary (malicious users and RPs) still fails to make one identity proof (or its transformation) 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.
- 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 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.
- 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}.

UPRESSO ensures the identification by binding the identity proof with PID_U in the form of $ID_{RP}^{ID_U}$, instead of a random string generated by the IdP. 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.

The design of UPRESSO makes it immune to some existing known attacks (e.g., CSRF, 307 Redirect, IdP Mix-Up [16] and Man-in-middle attack) 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 [16] 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. In the IdP Mix-up attack, the adversary works as the IdP to collect the makes `access token` and `authorization code` (identity proof in OAuth 2.0) from the victim RP. Same as OIDC, UPRESSO includes the `issuer` in the identity proof (protected by the SK_{ID}), avoiding the victim RP to send the sensitive information to the IdP. The user established the TLS connection with RP and IdP, which avoids the Man-in-middle attack.

VII. Implementation And 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 no less security strength than RSA-2048 [34].

The IdP is implemented based on MITREid Connect [35]. an open-source OIDC Java implementation certificated by the OpenID Foundation [36]. Alghough, OIDC standard specifies that RP's identifier should be generated by IdP in the dynamic registration, MITREid Connect allows the user to provide a candidate RP identifier to the IdP who checks the uniqueness, which simplifies the implementation of UPRESSO. 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 [37]. The Chrome extension clears the `referer` in the HTTP header, to avoid the RPs' URL leaked to the IdP. 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. The user's login request implemented by SDK contains Step 2-3, 5-7 and 11-12 in Figure 3; while identity proof parsing contains Step 18 in Figure 3.

B. 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.

Environment. 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.

Performance. We have measured the processing time for 1000 login flows, and the the results is demonstrated in Figure 4. The average time is 208 ms, 113 ms and 308 ms for UPRESSO, MITREid Connect and SPRESSO respectively.

For better comparison, we further divide a SSO login flow into 4 phases, which : 1. **authentication request initiation** (Steps 1-14 in Figure 3), 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 15 in Figure 3)**, denoting the construction of identity proof at the IdP (excluding the user authentication); 3. **identity proof transmitting** (Steps 16-17 in Figure 3), for transmitting the proof from the IdP to the RP with the

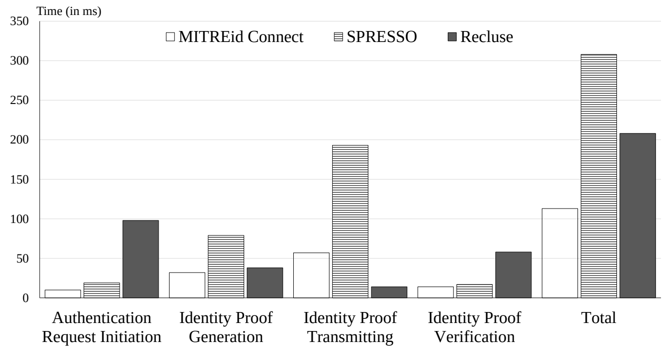

Fig. 4: The Evaluation.

user's help; and 4. **identity proof verification** (Steps 18 in Figure 3), for the RP verifying and parsing the proof for the user's $Account$.

In the authentication request initiation, MITREid Connect requires the shortest time (10 ms); SPRESSO needs 19 ms for RP to obtain the IdP's public information and encrypt its domain; UPRESSO needs 98 ms, for the $PRPID$ calculation (1 modular exponentiation at the user and 2 at the RP) and dynamic registration.

For identity proof generation, MITREid Connect needs 32 ms (information constructing and signing); UPRESSO needs an extra 6 ms for the generation of $PPID$; SPRESSO requires 71 ms for a different format of identity proof, which is longer than others as the processing in SPRESSO is implemented with JavaScript while the others are using Java and it costs more time for signature generation by JavaScript because of the programming language feature.

For identity proof transmitting, IdP in 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; the total transmitting requires 57 ms. In UPRESSO, a chrome extension relays the identity proof from the IdP to RP, which needs 14 ms. 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, which need about 193 ms.

In identity proof verification, the RP in MITREid Connect needs 14 ms for verifying the signature, SPRESSO requires 17 ms for a systematic decryption and signature verification, while UPRESSO needs 58 ms for calculation of $Account$ and signature verification.

VIII. Discussion

In this section, we provide some discussion about UPRESSO.

Support of 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. While the RP's identifier and secret are issued by the IdP in the dynamic registration described above.

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. 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, the code could be provided by IdP, which contains about only 300 lines of code and as it runs in the browser, IdP cannot modify or monitor the code without prior inserted malicious code. 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 2 minutes and during 2 minutes there are 10 million requests for dynamic registration. So IdP need to offer about 1 GB storage for dynamic registration. The extra cost of storage can be ignored.

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 is 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 3 in Figure 3, $Cert_{RP}$ 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.

Identity injection by malicious IdP. It has been discussed in [6] 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. SPRESSO requires that user should input her email at RP to avoid the identity injection, which is also available in UPRESSO by adding the extra user name (defined by user for each RP) input.

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 [38], and Central Authentication Service (CAS) [39] is widely adopted by Java applications. Kerberos [40], 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.

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 [41].

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. [19] performed a traffic analysis/manipulation on SSO implementations provided by Google and Facebook; [14], [21], [23] 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. [20] exploited the vulnerability at the user agent, to transmit the identity proof to the malicious RP. (2) The integrity is broken [14], [18], [19], [21], [23], [24], [26] in the implementations of SAML, OAuth and OIDC. For example, [18] 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 [14], [21], [23]; ID spoofing and key confusion make the identity proof issued by the adversary be accepted by the victim RPs [24], [26]. (3) The binding is also broken [14], [21], [23], [27], as the RP may misuse the bearer token as the identity proof [14], [21], [23], and IdP may not bind the refresh/access token with RP which allows the refresh/access token injection [27]. Cao et al. [42] 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.

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) [22], [25]. 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 [25]; the lack of authentication between mobile applications may also make the identity proof (or index) be leaked to the malicious applications [22]. Various automatic tester were proposed to analyze the mobile SSO systems [14], [22], [25], [28], [29], for the traditional vulnerabilities (e.g., inadequate transmission protection [22], token replacement [29]) and new ones in mobile platforms (webview [25], application logic error [28]).

The comprehensive formal security Analysis were performed on SAML, OAuth and OIDC. Armando et al. [15] 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. [16], [17] 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. [16], [17] 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. [43] 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.

Privacy is the another concern of SSO systems. As suggested in NIST SP800-63C [2], 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. OAuth and OIDC provide the user notification to achieve the user's control on its private information [13], [23]. The pairwise user identifier is proposed to avoid the identity linkage performed by colluded RPs in SAML and OIDC [7], [44]. In SPRESSO [6] and

BrowserID [9] (adopted in Persona [5] and its new version Firefox Accounts [45]), 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 colluded RPs. Fett et al. [9], [46] performed a formal analysis on the implementation of BrowserID and found that IdP may still know which RP is accessed by the user.

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 [47]. In 2013, the notion of anonymous single sign-on is formalized [48]. Then, the various cryptographic primitives, e.g., group signatures and zero-knowledge proof, are adopted to build anonymous SSO scheme [48], [49].

X. Conclusion

In this paper, we, for the first time, propose UPRESSO to preserve the users' privacy from the curious IdP and colluded 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 colluded 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.

References

[1] "The top 500 sites on the web," https://www.alexa.com/topsites, Accessed July 30, 2019.

[2] Paul A Grassi, M Garcia, and J Fenton, "Draft nist special publication 800-63c federation and assertions," *National Institute of Standards and Technology, Los Altos, CA*, 2017.

[3] Sydney Li and Jason Kelley, "Google screenwise: An unwise trade of all your privacy for cash," https://www.eff.org/deeplinks/2019/02/google-screenwise-unwise-trade-all-your-privacy-cash, Accessed July 20, 2019.

[4] Bennett Cyphers and Jason Kelley, "What we should learn from "facebook research"," https://www.eff.org/deeplinks/2019/01/what-we-should-learn-facebook-research, Accessed July 20, 2019.

[5] Mozilla Developer Network (MDN), "Persona," https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona.

[6] Daniel Fett, Ralf Küsters, and Guido Schmitz, "SPRESSO: A secure, privacy-respecting single sign-on system for the web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA*, 2015, pp. 1358–1369.

[7] Nat Sakimura, John Bradley, Mike Jones, Breno de Medeiros, and Chuck Mortimore, "Openid connect core 1.0 incorporating errata set 1," *The OpenID Foundation, specification*, 2014.

[8] Thomas Hardjono and Scott Cantor, "SAML v2.0 subject identifier attributes profile version 1.0," *OASIS standard*, 2019.

[9] Daniel Fett, Ralf Küsters, and Guido Schmitz, "Analyzing the browserid SSO system with primary identity providers using an expressive model of the web," in *20th European Symposium on Research in Computer Security (ESORICS)*, 2015, pp. 43–65.

[10] Nat Sakimura, John Bradley, Mike Jones, Breno de Medeiros, and Chuck Mortimore, "Openid connect dynamic client registration 1.0 incorporating errata set 1," *The OpenID Foundation, specification*, 2014.

[11] Dick Hardt, "The oauth 2.0 authorization framework," *RFC*, vol. 6749, pp. 1–76, 2012.

[12] Michael B. Jones and Dick Hardt, "The oauth 2.0 authorization framework: Bearer token usage," *RFC*, vol. 6750, pp. 1–18, 2012.

[13] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA*, 2014, pp. 892–903.

[14] Hui Wang, Yuanyuan Zhang, Juanru Li, and Dawu Gu, "The achilles heel of oauth: A multi-platform study of oauth-based authentication," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA*, 2016, pp. 167–176.

[15] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps," in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA*, 2008, pp. 1–10.

[16] Daniel Fett, Ralf Küsters, and Guido Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria*, 2016, pp. 1204–1215.

[17] Daniel Fett, Ralf Küsters, and Guido Schmitz, "The web SSO standard openid connect: In-depth formal security analysis and security guidelines," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA*, 2017, pp. 189–202.

[18] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen, "On breaking SAML: Be whoever you want to be," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA*, 2012, pp. 397–412.

[19] Rui Wang, Shuo Chen, and XiaoFeng Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *IEEE Symposium on Security and Privacy, SP 2012, San Francisco, California, USA*, 2012, pp. 365–379.

[20] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti, "An authentication flaw in browser-based single sign-on protocols: Impact and remediations," *Computers & Security*, vol. 33, pp. 41–58, 2013.

[21] Yuchen Zhou and David Evans, "SSOScan: Automated testing of web applications for single sign-on vulnerabilities," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA*, 2014, pp. 495–510.

[22] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu, "Vulnerability assessment of oauth implementations in android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA*, 2015, pp. 61–70.

[23] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu, "Model-based security testing: An empirical study on oauth 2.0 implementations," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China*, 2016, pp. 651–662.

[24] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk, "Do not trust me: Using malicious idps for analyzing and attacking single sign-on," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany*, 2016, pp. 321–336.

[25] Fadi Mohsen and Mohamed Shehab, "Hardening the oauth-webview implementations in android applications by re-factoring the chromium library," in *2nd IEEE International Conference on Collaboration and Internet Computing, CIC 2016, Pittsburgh, PA, USA*, 2016, pp. 196–205.

[26] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich, "Sok: Single sign-on security - an evaluation of openid connect," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France*, 2017, pp. 251–266.

[27] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang, "Vetting single sign-on SDK implementations via symbolic reasoning," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA*, 2018, pp. 1459–1474.

[28] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi, "Breaking and fixing mobile app authentication with oauth2.0-based protocols," in *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, Proceedings*, 2017, pp. 313–335.

[29] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau, "MoSSOT: An automated blackbox tester for single sign-on vulnerabilities in mobile applications," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand*, 2019, pp. 269–282.

[30] Xiaoyun Wang, Guangwu Xu, Mingqiang Wang, and Xianmeng Meng, *Mathematical foundations of public key cryptography*, CRC Press, 2015.

[31] Victor Shoup, "Searching for primitive roots in finite fields," *Mathematics of Computation*, vol. 58, no. 197, pp. 369–380, 1992.

[32] Wang Yuan, "On the least primitive root of a prime," in *Selected Papers Of Wang Yuan*, pp. 109–122. World Scientific, 2005.

[33] Whitfield Diffie and Martin E. Hellman, "New directions in cryptography," *IEEE Trans. Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[34] Elaine Barker, "Recommendation for key management part 1: General (revision 4)," *NIST special publication*, vol. 800, no. 57, pp. 1–160, 2016.

[35] "MITREid connect /openid-connect-java-spring-server," https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server, Accessed August 20, 2019.

[36] "Openid foundation," https://openid.net/certification/, Accessed August 20, 2019.

[37] "jsrsasign," https://kjur.github.io/jsrsasign/, Accessed August 20, 2019.

[38] "The shibboleth project," https://www.shibboleth.net, Accessed July 30, 2019.

[39] Pascal Aubry, Vincent Mathieu, and Julien Marchal, "ESUP-portail: Open source single sign-on with cas (central authentication service)," *Proc. of EUNIS04–IT Innovation in a Changing World*, pp. 172–178, 2004.

[40] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller, "Kerberos: An authentication service for open network systems," in *Proceedings of the USENIX Winter Conference. Dallas, Texas, USA*, 1988, pp. 191–202.

[41] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis, "O single sign-off, where art thou? An empirical analysis of single sign-on account hijacking and session management on the web," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA*, 2018, pp. 1475–1492.

[42] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Krügel, Giovanni Vigna, and Yan Chen, "Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden. Proceedings*, 2014, pp. 276–298.

[43] Quanqi Ye, Guangdong Bai, Kailong Wang, and Jin Song Dong, "Formal analysis of a single sign-on protocol implementation for android," in *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia*, 2015, pp. 90–99.

[44] John Hughes, Scott Cantor, Jeff Hodges, Frederick Hirsch, Prateek Mishra, Rob Philpott, and Eve Maler, "Profiles for the oasis security assertion markup language (SAML) v2. 0," *OASIS standard*, 2005, Accessed August 20, 2019.

[45] "About firefox accounts," https://mozilla.github.io/application-services/docs/accounts/welcome.html, Accessed August 20, 2019.

[46] Daniel Fett, Ralf Küsters, and Guido Schmitz, "An expressive model for the web infrastructure: Definition and application to the browserid SSO system," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*, 2014, pp. 673–688.

[47] Kalid Elmufti, Dasun Weerasinghe, Muttukrishnan Rajarajan, and Veselin Rakocevic, "Anonymous authentication for mobile single sign-on to protect user privacy," *IJMC*, vol. 6, no. 6, pp. 760–769, 2008.

[48] Jingquan Wang, Guilin Wang, and Willy Susilo, "Anonymous single sign-on schemes transformed from group signatures," in *2013 5th International Conference on Intelligent Networking and Collaborative Systems, Xi'an city, Shaanxi province, China*, 2013, pp. 560–567.

[49] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer, "Anonymous single-sign-on for n designated services with traceability," in *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, Proceedings, Part I*, 2018, pp. 470–490.