UPPRESSO: An Unlinkable Privacy-PREserving Single Sign-On System

***Abstract*—As a widely adopted identity management and authentication mechanism in today's Internet, single sign-on (SSO) allows a user to maintain only the credential for the identity provider (IdP), instead of one credential for each relying party (RP), which shifts the burden of user authentication from RPs to the IdP. However, SSO introduces new privacy leakage threats, since (*a*) a curious IdP could track *all* the RPs a user has visited, and (*b*) collusive RPs could learn a user's online profile by linking her identifiers and activities across multiple RPs. Several privacy-preserving SSO solutions have been proposed to defend against either the curious IdP or collusive RPs, however, none of them can address both privacy leakage threats at the same time.**

In this paper, we propose a privacy-preserving SSO system, called *UPPRESSO*, to protect a user's login traces against both the curious IdP and collusive RPs. We first formally analyze the privacy dilemma between SSO security requirements and the new privacy requirements, and convert the SSO privacy problem into an identifier-transformation problem. Then, we design a novel *transformed RP designation* scheme to transform the identifier of the RP, to which the user requests to log in, into a privacy-preserving pseudo-identifier (PID_{RP}) through the cooperation between the user and the RP. Our *trapdoor user identification* scheme allows the RP to obtain a trapdoor from the transformation process and use it to derive a unique account of the user at that RP from her privacy-preserving pseudo-identifier (PID_U) generated by the IdP. The login process of UPPRESSO follows the service pattern of OpenID Connect (OIDC), a widely deployed SSO system, with minimum modifications. And the system is platform independent. Our analysis shows UPPRESSO provides a comprehensive privacy protection while achieving the same security guarantees of OIDC.

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

I. Introduction

As a widely deployed identity management and authentication mechanism in the current Internet, single sign-on (SSO) systems such as OpenID Connect [1], OAuth [2] and SAML [3] allow a user to log in to a website, called the *relying party* (RP), using the account registered at another website, called the *identity provider* (IdP). The RPs delegate user authentication to a trusted IdP, who generates *identity proofs* for her visits to these RPs. Thus, the user only needs to remember one credential for the IdP, instead of maintaining different credentials for different RPs. SSO has been widely integrated with many application services. For example, we find that 80% of the Alexa Top-100 websites support SSO [4], and the analysis on the Alexa Top-1M websites identifies 6.30% with the SSO support [5]. Meanwhile, many email and social network providers (such as Google, Facebook, Twitter, etc.) are serving the IdP roles in the Internet.

However, SSO systems have been continuously found vulnerable and insecure [6]–[15]. Moreover, the adoption of SSO raises a public concern about user privacy [16]–[19], that is whether an adversary is able to track to which RP(s) the user has logged in. Unfortunately, almost all the existing SSO protocols leak user privacy in different ways. Take a widely used SSO protocol OpenID Connect (OIDC) as an example. As shown in Fig. 1, the login process starts when a user sends a login request to the RP, who then constructs a request for identity proof with its identity and redirects the request to the IdP. After authenticating the user, the IdP generates an identify proof with the user's and RP's identities, which is returned to the user and forwarded to the RP. Finally, the RP verifies the identity proof to decide if the user is allowed to log in. In such login instances, by design, an IdP can always see when and where its users log in, in order to generate the identity proof. As a result, a curious IdP can always discover the RPs that a target user has visited over time. This data can be further analyzed to profile users' online activities. Thus, we call this privacy attack ***IdP-based login tracing***, which has also been reported by previous research [18], [19]. Similarly, by design, the RPs can learn users' identities from the identify proofs. If the IdP binds an unique or relevant user identifier(s) to identity proofs generated for the same user but different RPs [20], [21], these RPs can collude to correlate the identifier(s) with the user's identity. We denote this privacy risk as ***RP-based identity linkage***, which allows the adversaries to not only track the user's online activities but also associate her attributes across multiple RPs by linking her login requests [16].

As SSO becomes a popular safeguard for various privacy-sensitive web services, the privacy concern is considered more prominent and severe than it was in the past. On one hand, privacy-savvy users may provide no or few personal information to web applications to avoid user tracking or profiling. On the other hand, the use of popular SSO services such as Google Account opens a door for IdPs and application providers to recover users' online traces and profiles, which makes users' privacy protection effort in vain. Several large IdPs, especially the social IdPs, are known to be interested in collecting users' online behavioral data for various purposes (e.g., Screenwise Meter [22] and Onavo [23]). Serving the IdP role makes it possible for them to collect such information. Meanwhile, service providers hosting multiple web applications take an advantaged position to correlate users' multiple logins at different RPs through internal information integration. Finally, privacy-preserving record linkage [24] and private set intersection [25] technologies allow multiple RPs to share data without violating their clients' privacy, which pave the path for cross-organizational RP-based identity linkage.

Several solutions have been proposed to protect user privacy in SSO login [16]–[19]. However, to the best of our knowledge, none of them provides a comprehensive protection to defend against IdP-based login tracing and RP-based identity linkage *at the same time*. For example, as recommended by NIST [17] and specified in several SSO protocols [1], [26], pairwise pseudonymous identifier (PPID) is generated by the IdP to identify a user to an RP, which cannot be correlated with the user's PPID at another RP. Thus, collusive RPs cannot link a user's logins from her PPIDs. However, PPID-based approaches cannot prevent IdP-based login tracing, since the IdP needs to know which RP the user visits in order to generate the correct identify proof. On the contrary, BrowserID [18] and SPRESSO [19] were proposed to defend against IdP-based login tracing. However, both solutions are vulnerable to RP-based identity linkage. In BrowserID (and its prototypes known as Mozilla Persona [27] and Firefox Accounts [21]), the IdP does not know the identity of the requesting RP. Instead, it generates a special "identity proof" to bind the user's unique identifier (e.g., email address) to a public key, so that the user can sign another subsidiary identity proof to bind her identity with the RP's identity and send both identity proofs to the RP. Obviously, when a user logs in to different RPs, the RPs can extract a same user identifier from different identity proofs and correlate these logins. In SPRESSO, the RP creates a one-time pseudo-identifier in each login. Then, the IdP generates an identity proof binding this pseudo-identifier and the user's identity (i.e., email address). Similarly, the RPs can correlate a user's logins using her unique identifier in the identity proofs.

Unfortunately, the techniques proposed by previous research cannot be directly integrated to address the two major types of privacy risks in SSO at the same time. In fact, it requires a non-trivial redesign of the SSO system to defend against IdP-based login tracing and RP-based identity linkage while providing a secure and compatible SSO service. In this paper, we first conceptualize the privacy problem in SSO as *an identifier transformation problem* and explain the reasons that limit existing solutions from fully protecting user privacy against curious IdPs and collusive RPs. Based on our analysis, we propose an Unlinkable Privacy-PREserving Single Sign-On (UPPRESSO) system to provide a comprehensive protection against both types of privacy attacks.

UPPRESSO designs three one-way identifier-transformation functions based on the discrete logarithm problem. Using the one-way trapdoor function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}(ID_{RP}, T)$, the RP converts its identity ID_{RP} into a privacy-preserving pseudo-identifier PID_{RP} based on a randomly selected trapdoor T. Similarly, the IdP uses the one-way function $\mathcal{F}_{ID_U \mapsto PID_U}(ID_U, PID_{RP})$ to generate a privacy-preserving pseudo-identifier PID_U for the user based on her identity ID_U and PID_{RP}. Finally, using a special identifier-transformation function $\mathcal{F}_{PID_U \mapsto Account}(PID_U, PID_{RP}, T)$, the RP is able to map all the different privacy-preserving pseudo-identifiers of a user, which are created in her different login sessions to that RP, to a same $Account$ that identifies the user to the RP. The three identifier-transformation functions work cooperatively to ensure: (*a*) when a user logs in to an RP multiple times, the RP can always map PID_Us to a unique $Account$ without knowing the user's identity ID_U; moreover, when a user logs in to multiple RPs, (*b*) a curious IdP learns nothing about the identities of these RPs from PID_{RP}s, and (*c*) collusive RPs cannot link PID_Us to a particular user (*d*) nor correlate $Account$s of a same user at different RPs. We summarize our contributions as follows.

- We are among the first to conceptualize the privacy problem in SSO as an identifier-transformation problem and analyze the strengths and limitations of existing SSO privacy protection solutions.
- We propose a comprehensive solution to hide the users' login traces from curious IdPs and collusive RPs. To the best of our knowledge, UPPRESSO is the first SSO system that secures SSO services against IdP-based login tracing and RP-based identity linkage.
- We analyze the security of UPPRESSO based on a formal model of the web infrastructure and formally prove that it provides satisfying security and privacy properties.
- We implement a prototype of UPPRESSO based on an open-source implementation of OIDC, which requires only small modifications to support three identifier-transformation functions for privacy protections. Thus, UPPRESSO is compatible with existing SSO systems. Moreover, our prototype leverages HTML 5 features in the implementation so that it can be used across platforms (e.g., PCs, smart phones and other devices).
- We compare the performance of the UPPRESSO prototype with the state-of-the-art SSO systems (i.e., OIDC [1] and SPRESSO [19]) and demonstrate its efficiency.

The rest of the paper is organized as follows. We first introduce the background and preliminaries in Section II. Then, we describe the identifier-transformation-based approach and the threat model in Sections III and IV. Section V presents the details of our UPPRESSO design, followed by a formal analysis of its security and privacy in Section VI. We explain the implementation specifics and experiment evaluation in Section VII, discuss the extensions and related works in Section VIII and IX, and conclude our work in Section X.

II. Background and Preliminaries

UPPRESSO is designed to be compatible with OpenID Connect (OIDC) and provide privacy protections based on the discrete logarithm problem. Next, we briefly introduce OIDC and the discrete logarithm problem.

A. OpenID Connect (OIDC)

OIDC is one of the most popular SSO protocols [1]. It involves three entities, i.e., *users*, the *identity provider (IdP)*, and *relying parties (RPs)*. Users and RPs register at the IdP with identifiers and other necessary information such as credentials and RP endpoints (e.g., the URLs to receive identity proofs). The IdP is assumed to maintain these attributes securely.

OIDC Implicit Flow. OIDC supports three types of user login flows: *implicit flow*, *authorization code flow* and *hybrid flow* (i.e., a mix-up of the previous two). UPPRESSO is compatible with all three flows. For brevity, we will present our design and implementation on top of the OIDC implicit flow in the rest of the paper and discuss the extension to support the authorization code flow in Section VIII.

As shown in Figure 1, first, the user initiates a login request to an RP. Then, the RP constructs an identity proof request

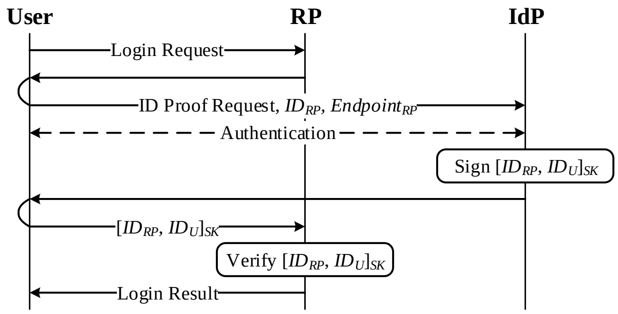

Fig. 1: The implicit flow of OIDC.

with its identifier, an endpoint to receive the identity proof and a scope of requested user attributes, and sends the request to the user who will redirect it to the IdP. If the user has not been authenticated yet, the IdP initiates an authentication process to authenticate the user based on her identity and credential. If privacy-preserving pseudo-identifier is used, this process also involves mapping ID_U to PID_U based on ID_{RP}. Once successfully authenticating the user, the IdP generates an identity proof (called *id token*) and returns it to the RP endpoint through user redirection. The id token contains a user identifier (ID_U or PID_U), an RP identifier (ID_{RP}), the issuer, a validity period, the requested user attributes, etc. If the RP's endpoint has not been registered at the IdP, the IdP will return a warning to notify the user about potential identity proof leakage. Besides redirecting the messages between the RP and the IdP, the user also checks if the RP is permitted to obtain the user attributes in the identify proof. Usually, the redirection and checking actions are handled by a user-controlled software, called *user agent* (e.g., browser). Finally, the RP verifies the received identity proof and makes the authentication decision.

RP Dynamic Registration. OIDC also supports *RP dynamic registration* [28]. When an RP first registers at an IdP, it obtains a registration token with which the RP can update its information (e.g., endpoints) with the IdP in a later time. After a successful dynamic registration, the RP obtains a new ID_{RP} from the IdP. UPPRESSO leverages this function and slightly modifies the dynamic registration process to implement the PID_{RP} *registration* process (see details in Section V.C), which allows an RP to generate different privacy-preserving RP identifiers and register them with the IdP.

B. Discrete Logarithm Problem

Based on the discrete logarithm problem, UPPRESSO designs the identifier-transformation functions. Here, we briefly review the discrete logarithm problem. For the finite field $GF(p)$ where p is a large prime, a number g is called a generator of order q, if it constructs a cyclic group of q elements by calculating $y = g^x \ mod \ p$. And x is called the discrete logarithm of y modulo p. Given a large prime p, a generator g and a number y, it is computationally infeasible to solve the discrete logarithm (i.e., x) of y [29], which is called the discrete logarithm problem. The hardness of solving discrete logarithms is utilized to design several secure cryptographic primitives, including Diffie-Hellman key exchange and the digital signature algorithm (DSA).

TABLE I: The notations used in UPPRESSO.

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

III. The Privacy Dilemma and UPPRESSO Overview

Next, we overview the required security and privacy properties of an SSO system. Then, we conceptualize the SSO privacy problem as an identifier-transformation problem and explain the privacy dilemma behind existing solutions. Finally, we present the design goals of UPPRESSO. We list the notations used in the discussion in Table I for reference.

A. Security Properties of SSO

The primary goal of SSO services is to support secure user authentication [19], which ensures that a legitimate user can always log in to an honest RP under her account. To achieve this, the identity proof generated by the IdP should explicitly specify the user who is authenticated by the IdP (i.e., ***user identification***) and the RP to which the user requests to log in (i.e., ***RP designation***). To provide a continuous service, the user identification property also requires an RP to be able to recognize a user and correlate her multiple logins by a unique identifier (or account). Moreover, the identify proof generated by the IdP should be transmitted only to the dedicated RP (through the user) (i.e., ***confidentiality***) and should not be modified or forged (i.e., ***integrity***). We summarize these four security properties from theoretical analysis of SSO designs [30]–[32] and practical attacks [6]–[15], [33]–[40].

Many attacks exploit vulnerabilities in SSO design and implementation to break at least one of the four security properties. The adversary mainly aims to log in to an honest RP as a victim user (called *impersonation attacks*) or allure a victim user to log in to an honest RP under the attacker's account (called *identity injection attacks*). For example, Friendcaster used to accept every received identity proof (i.e., a violation of RP designation) [38]. So, a malicious RP can replay a received identity proof to Friendcaster and log in as the victim user. If identity proofs are leaked (i.e., a violation of confidentiality) [6], [7], [9]–[11], the adversary can directly impersonate the victim user. It was also reported that some RPs of Google ID SSO accepted user attributes that were not tied to the identity proof (i.e., a violation of integrity) [6]. This allows an adversary to insert arbitrary attributes (e.g., email address of the adversary or another user) into the identity proof for the victim user.

B. The Privacy Dilemma in SSO Identity Proofs

A secure SSO system should have *all* four security properties discussed above while preventing IdP-based login tracing and RP-based identity linkage privacy leakage. However, meeting the security and privacy requirements at the same time incurs a dilemma in the generation of identity proofs.

An identity proof contains identities/identifiers of a user and an RP, which is to tell the RP that this user has been authenticated by the IdP. Since the IdP always knows the identity of the user (denoted as ID_U), to prevent IdP-based login tracing, we should not reveal RP's long-term identity (denoted as ID_{RP}) to the IdP. Instead, we have to use a transitional pseudo-identifier (denoted as PID_{RP}) that is uniquely associated with the RP in the identity proof request to ensure RP designation. Each RP's PID_{RP}s in different login instances should be different. PID_{RP} can be generated by the user, the RP or together, but it should be computationally infeasible for the IdP to derive the ID_{RP} from a PID_{RP}.

Meanwhile, to prevent RP-based identity linkage, the IdP should not directly include ID_U in the identity proof. So, the IdP has to generate a transitional pseudo-identifier for the user (denoted as PID_U) and bind it to the identity proof. While PID_U should not disclose any information for the RP to derive ID_U, it should also allow the RP to recognize the user and distinguish her from other users in the RP, which means an RP should be able to correlate a user's different PID_Us in different login instances, for example by mapping them to a unique user account (denoted as $Account$) at the RP, to ensure user identification. However, two or more RPs should not be able to correlate a user's $Account$s at different RPs to infer that they belong to the same user.

We illustrate the relationships among the identities, pseudo-identifiers and the identity proofs in Figure 2. The red and green blocks represent long-term identities and one-time pseudo-identifiers respectively, and the arrows denote how the pseudo-identifiers are obtained. To comprehensively protect user privacy, the identity proof should only use one-time pseudo-identifiers of the user and the RP, where PID_U and PID_{RP} should satisfy the above requirements. This cause a ***dilemma*** for the IdP: given a user (ID_U) and an unknown RP (PID_{RP}), the IdP is expected to generate a pseudo-identifier (PID_U), which is correlated with a long-term account that uniquely identifies the user at that RP, *without knowing anything about the RP's identity nor the user account at the RP*.

To solve the dilemma, we should provide the IdP some information related to the user's $Account$ at the RP to assist the generation of PID_U, so that PID_U can be correctly correlated with the $Account$. Meanwhile, such information should not provide any additional knowledge for the IdP to derive the RP's identity, or for two RPs to correlate two $Account$s belonging to the same user. Therefore, the privacy protection problem can be converted into an identifier-transformation problem, which aims to design three identifier-transformation functions $\mathcal{F}_{ID_U \mapsto PID_U}$, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$, and $\mathcal{F}_{PID_U \mapsto Account}$ to compute PID_U, PID_{RP} and $Account$ that provide the above desired properties.

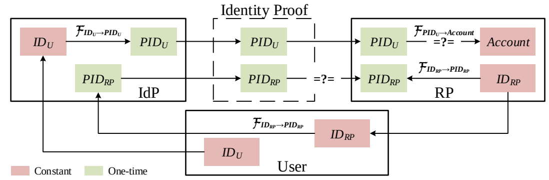

Fig. 2: Identifier transformations in privacy-preserving SSO.

C. The Identifier-transformation Framework of UPPRESSO

To achieve this goal, UPPRESSO constructs three transformation functions in an integrated way to support *transformed RP designation* and *trapdoor user identification*, where (a) different PID_Us and PID_{RP}s are dynamically generated in different logins; (b) in each login session, PID_{RP} is used to assists the generation of PID_U, which helps to link PID_U to $Account$ using the trapdoor of this session.

Transformed RP designation. To prevent IdP-based login tracing, the RP includes a PID_{RP} dynamically transformed from ID_{RP}, instead of ID_{RP}, in the identity proof request. UPPRESSO designs a novel trapdoor-based transformation function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}(ID_{RP}, T)$ to compute PID_{RP} based on ID_{RP} and a random trapdoor T, which is dynamically negotiated between the user and the RP in each login. Then, the user assists the RP to register PID_{RP} at the IdP through OIDC dynamic registration. When an RP receives an identity proof, it verifies if the enclosed PID_{RP} is transformed from its ID_{RP} and the trapdoor of this session.

Trapdoor user identification. To prevent RP-based identity linkage, we design a transformation function $\mathcal{F}_{ID_U \mapsto PID_U}(ID_U, PID_{RP})$ for the IdP to generate PID_U. When an RP receives an identity proof, another transformation function $\mathcal{F}_{PID_U \mapsto Account}(PID_U, PID_{RP}, T)$ is designed to help the RP to derive the $Account$ from PID_U and PID_{RP} using the trapdoor it holds. Intuitively, the trapdoor T plays a role in the generations of PID_{RP} and PID_U, directly or indirectly.

D. Existing Privacy-Preserving SSO Solutions

We map three existing privacy-preserving SSO approaches (PPID [1], BrowserID [18] and SPRESSO [19]) to the identifier transformation framework in Figure 2 and summarize their potential privacy issues in Table II. It is worth noting that when $PID_U = ID_U$ and $PID_{RP} = ID_{RP}$, this framework depicts the basic SSO services with no privacy protection.

In PPID approaches, the IdP generates different PID_Us for a user to log in to different RPs and maintains deterministic one-to-many mappings from ID_U to PID_Us. Therefore, they can prevent RP-based identity linkage. At each RP, a user is identified by a same PID_U (i.e., $Account = PID_U$), which ensures user identification. However, since ID_{RP} is directly used in identity proofs (i.e., $PID_{RP} = ID_{RP}$), these approaches are vulnerable to IdP-based login tracing.

In SPRESSO, the RP generates PID_{RP} by encrypting ID_{RP} padded with a nonce for each login session and forwards PID_{RP} to the IdP. With the corresponding nonce, the RP can verify PID_{RP} in the identity proof to ensure

TABLE II: Identifier-transformation in privacy-preserving SSO.

Solution	PID_U	PID_{RP}	$Account$
PPID	$\mathcal{F}(ID_U, ID_{RP})$	ID_{RP}	PID_U
SPRESSO	ID_U	$Enc(ID_{RP}\|nonce)$	ID_U
BrowserID†	ID_U	$\perp$	ID_U
UPPRESSO	$\mathcal{F}(ID_U, PID_{RP})$	$\mathcal{F}(ID_{RP}, T)$	$\mathcal{F}(PID_U, T)$

†: BrowserID binds null PID_{RP} in the identity proofs by the IdP, but ID_{RP} is bound in the *subsidiary* identity proof signed by the user.

RP designation, while hiding ID_{RP} from the IdP to defend against IdP-based login tracing. However, a same ID_U is used to generate identity proofs for a same user, no matter which RPs she requests to log in. So, SPRESSO is vulnerable to RP-based identity linkage, since different RPs can correlate login requests of a same user by ID_U (i.e., $Account = ID_U$).

Identity proofs in BrowserID include ID_U but no RP information (i.e., $PID_{RP} = \perp$), therefore, it directly prevents IdP-based login tracing. To ensure RP designation, BrowserID requires the user to append a *subsidiary* identity proof and sign it, where the identity proof signed by the IdP authorizes the user to sign the subsidiary identity proof. Obviously, ID_U is tied to a pair of identity proof and subsidiary identity proof. Similarly, a user's login requests to different RPs can be linked by ID_U (i.e., $Account = ID_U$), which makes BrowserID vulnerable to RP-based identity linkage.

None of the three approaches can defend against IdP-based login tracing and RP-based identity linkage at the same time. This is because in each approach, three transformation functions $\mathcal{F}_{ID_U \mapsto PID_U}$, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ and $\mathcal{F}_{PID_U \mapsto Account}$ are designed arbitrarily and function separately, which causes either $PID_{RP} = ID_{RP}$ or $Account = ID_U$.

IV. Threat Model and Assumptions

A. Threat Model

In UPPRESSO, we consider the IdP is curious-but-honest, while some users and RPs could be compromised by adversaries. Malicious users and RPs may behave arbitrarily or collude with each other, attempting to break the security and privacy guarantees for benign users.

Curious-but-honest IdP. A curious-but-honest IdP strictly follows the protocol, while being interested in learning user privacy. For example, it may store all the received messages to infer the relationship among ID_U, ID_{RP}, PID_U and PID_{RP} to trace a user's login activities at multiple RPs. We also assume the IdP is well-protected. For example, the IdP is trusted to maintain the private key for signing identity proofs and RP certificates, so, the adversaries cannot forge an identity proof or an RP certificate.

Malicious Users. We assume the adversary can control a set of users, for example by stealing users' credentials [41], [42] or directly registering sybil accounts at the IdP and RPs. They may impersonate a victim user at honest RPs, or trick a victim user to log in to an honest RP under the adversary's account. For example, a malicious user may modify, insert, drop or replay a message, or deviate arbitrarily from the specifications when processing ID_{RP}, PID_{RP} and identity proofs.

Malicious RPs. The adversary can also control a set of RPs, for example, by directly registering at the IdP as an RP or exploiting software vulnerabilities to compromise some RPs. The malicious RPs may behave arbitrarily to break security and privacy guarantees. To do so, a malicious RP may manipulate its PID_{RP} to trick the users to submit identity proofs generated for an honest RP to itself. or it may manipulate its PID_{RP} to affect the generation of PID_U and analyze the relationship between PID_U and $Account$.

Collusive Users and RPs. Malicious users and RPs may collude with each other to break the security and privacy guarantees. For example, acting as an RP, the adversary first lures a victim user to submit a valid identity proof to itself, and then logs in to the honest RPs as the victim user using this identity proof.

B. Assumptions

We also make a few assumptions about the information and implementation of the SSO system under study. First, we consider user attributes as distinctive and indistinctive attributes, where distinctive attributes contain identifiable information about a user such as telephone number, address, driver license, etc. We assume the RPs cannot obtain distinctive attributes in an SSO login, since a privacy-savvy user is less likely to permit the RPs to access such information, or even not register such information with the IdP at all. Thus, privacy leakage due to user re-identification is considered out of the scope of this work. Also, we focus only on privacy attacks enabled by SSO protocols, but not network attacks such as traffic analysis that can trace a user's logins at different RPs.

Secondly, we assume the user agent deployed at honest users is correctly implemented so that it can transmit messages to the dedicated receivers as expected. We also assume TLS is adopted to secure the communications between honest entities. Moreover, we assume the cryptographic algorithms (such as RSA and SHA-256) and building blocks (such as random number generators and the discrete logarithm problem) are correctly implemented.

V. The Design of UPPRESSO

Once we conceptualize the privacy problem into a identifier-transformation problem, the design of UPPRESSO is mainly about designing three identifier-transformation functions to generate pseudo-identifiers for the user and RP as well as link the user's pseudo-identifier to her account at an RP. In this section, we first present our design of these three functions to support *transformed RP designation* and *trapdoor user identification* properties, and then describe the details of the UPPRESSO system and its login flow.

A. Identifier-transformation Functions in UPPRESSO

We construct the three functions, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$, $\mathcal{F}_{ID_U \mapsto PID_U}$ and $\mathcal{F}_{PID_U \mapsto Account}$, based on the discrete logarithm problem with public parameters p, q, where p is a large prime defining the finite field $GF(p)$, and q is a prime factor of $(p-1)$. Without loss of generality, we assume the IdP assigns long-term identifiers ID_U to a user and ID_{RP} to an RP when they first register at the IdP. In particular, the IdP assigns a unique random number to each user as ID_U, where $1 < ID_U < q$, and ID_{RP} is a generator of order q in $GF(p)$.

The RP Identifier Transformation Function. In each login session, the user assists the RP to convert ID_{RP} into a pseudo-identifier PID_{RP}. In particular, the user selects a random number N_{RP} ($1 < N_{RP} < q$) and calculates PID_{RP} as:

$$\mathcal{F}_{ID_{RP} \mapsto PID_{RP}} : PID_{RP} = {ID_{RP}}^{N_U} \bmod p \qquad (1)$$

This transformation function $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ is a one-way function so that it is computationally infeasible for the IdP to derive ID_{RP} from PID_{RP} due to the discrete logarithm problem. Moreover, the nonce N_U ensures that: (a) PID_{RP} is a one-time pseudo-identifier of a login that is valid only for the identity proof generated in this login; and (b) PID_{RP} is dynamically generated in each login. When a user visits a same RP multiple times, different PID_{RP}s will be generated, which cannot be correlated or linked to a same RP.

The User Identifier Transformation Function. Now, the identity proof request to the IdP contains a user identity ID_U and a pseudo-identifier of the RP PID_{RP}. Therefore, the IdP can convert ID_U into a pseudo-identifier for the user as follows and use it in the identity proof:

$$\mathcal{F}_{ID_U \mapsto PID_U} : PID_U = {PID_{RP}}^{ID_U} \bmod p \qquad (2)$$

From Equations 1 and 2, we see that $PID_U = {ID_{RP}}^{N_U ID_U} \bmod p$. So, PID_U is a one-time pseudo-identifier that is valid only in one login session and one identity proof. The discrete logarithm problem ensures that the RP cannot derive ID_U from PID_U. Moreover, although the IdP does not know how the RP identifies the user (i.e. the user's $Account$ at the RP), involving PID_{RP} in the generation of PID_U indirectly links a user's one-time pseudo-identifier at the IdP (PID_U) to her long-term identifier at the RP ($Account$) through a trapdoor.

The User Account Transformation Function. We define the trapdoor T of each login session as $T = N_U^{-1} \bmod q$. As q is a prime number and $1 < N_U < q$, q is coprime to N_U. So, there always exists a T that satisfies $TN_U = 1 \bmod q$. Using this trapdoor, the RP can easily derive a unique account (denoted as A in the equation) for the user in each login session as:

$$\mathcal{F}_{PID_U \mapsto Account} : A = {PID_U}^T \bmod p \qquad (3)$$

From Equations 1, 2 and 3, we can further derive:

$$A = {PID_U}^T = {ID_{RP}}^{ID_U N_U N_U^{-1} \bmod q} = {ID_{RP}}^{ID_U} \bmod p$$

This means, when a user logs in to an RP multiple times, the RP can always derive the same $Account$ from different PID_Us to uniquely identity the user. However, the RP cannot derive ID_U from $Account$ due to the discrete logarithm problem. Finally, $Account$ provides no clue for different RPs to correlate the users.

With three identifier-transformation functions, UPRESSO supports two desirable properties discussed in Section III.C to satisfy *all* the security and privacy requirements of an SSO. ***(i) Transformed RP designation:*** using $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$, the user and RP cooperatively generate a dynamic PID_{RP} for each login. The identify proof request contains PID_{RP} instead of ID_{RP}, so, the RP can verify PID_{RP} is associated with ID_{RP} using the trapdoor but the IdP cannot tell to which RP the user attempts to log in. Also, since PID_{RP}s of a same RP are different in different login sessions, the IdP cannot even tell if a same RP is visited. Therefore, it prevents IdP-based login tracing. ***(ii) Trapdoor user identification:*** For each user, different PID_Us are generated by the IdP in different login sessions, no matter she requests to log in to a same RP multiple times or to different RPs. However, using $\mathcal{F}_{ID_U \mapsto PID_U}$ and $\mathcal{F}_{PID_U \mapsto Account}$, UPRESSO guarantees that an RP can always derive the unique $Account$ for each user using the dynamically generated PID_U and the corresponding trapdoor in each login session. Meanwhile, collusive RPs cannot link a user's PID_Us and $Account$s at different RPs, and therefore prevents RP-based identity linkage.

B. UPPRESSO Procedures

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

RP Initial Registration. Each RP calls an initial registration process once to obtain the necessary configurations from the IdP. In particular, an RP registers itself at the IdP to obtain a unique identifier ID_{RP} and the corresponding RP certificate $Cert_{RP}$ as follows: (i) The RP sends a registration request to the IdP, including the RP endpoint (e.g., URL) to receive identity proofs; (ii) The IdP generates a unique ID_{RP} and signs $[ID_{RP}, Endpoint_{RP}, *]$ using SK, where $*$ denotes supplementary information such as the RP's common name; then, the IdP returns $Cert_{RP} = [ID_{RP}, Endpoint_{RP}, *]_{SK}$ to the RP, where $[\cdot]_{SK}$ means the message is signed using SK; (iii) The RP verifies $Cert_{RP}$ using PK and accepts ID_{RP} and $Cert_{RP}$ if they are valid. Note that, in UPRESSO, ID_{RP} must be generated by the IdP but cannot be chosen by the RP.

User registration. UPPRESSO adopts a similar user registration process as the ones in other SSO systems. Each user registers once at the IdP to set up a unique user identifier ID_U and the corresponding user credential. ID_U can be chosen by the user or the IdP, as long as it is unique for each user.

SSO Login. An SSO login procedure is launched when a user requests to log in to an RP, which calls three identifier-transformation functions following the login flow as shown in Figure 3. It consists of five phases, namely scripts downloading, RP identifier transformation, PID_{RP} registration, identity proof generation and $Account$ calculation.

1. Scripts Downloading. The scripts downloading phase is for the user's browser to download the scripts from the RP and IdP servers. The browser and two scripts work together to play the user agent role.

1.1 The user visits the RP's script site to download the RP script.
1.2 The RP script opens a new window in the browser to visit the login path at the RP server.
1.3 The visit is redirected to the IdP's script site.

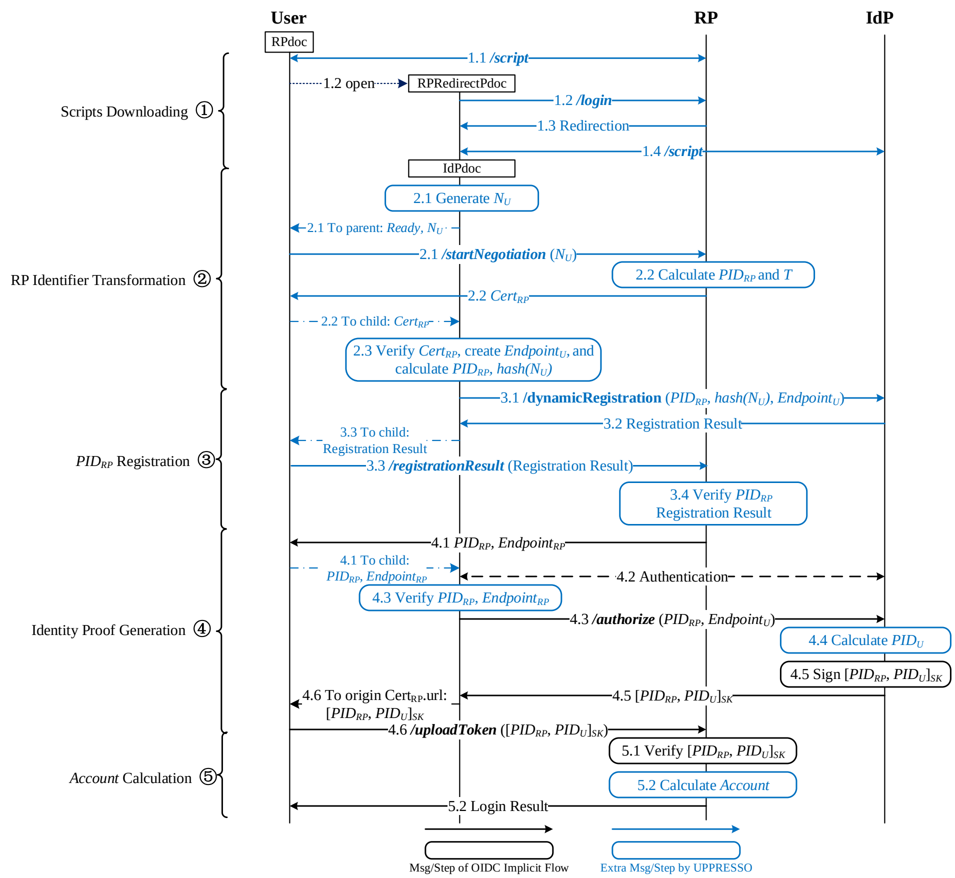

Fig. 3: The flow of a user login in UPPRESSO.

1.4 The new window visits the IdP's script site and downloads the IdP script.

2. RP Identifier Transformation. The user and RP cooperate to generate $PID_{RP} = {ID_{RP}}^{N_U} \bmod p$. To hide the RP's endpoint from the IdP, the user needs to create a new endpoint to replace the real endpoint of the RP.

2.1 The IdP script chooses a random N_U $(1 < N_U < q)$ and sends it to the RP script through postMessage. Then, the RP script sends N_U to the RP server.
2.2 The RP verifies $N_U \neq 0 \bmod q$, calculates PID_{RP} and derives the trapdoor $T = (N_U N_{RP})^{-1} \bmod q$. To acknowledge the negotiation of PID_{RP}, the RP replies with $Cert_{RP}$, which is transmitted from the RP script to the IdP script through postMessage.
2.3 The IdP script verifies $Cert_{RP}$, extracts ID_{RP} from $Cert_{RP}$ and calculates $PID_{RP} = {ID_{RP}}^{N_U} \bmod p$ and $nonce = hash(N_U)$. It also creates a one-time endpoint for the RP. If $Cert_{RP}$ is invalid, the user halts the negotiation.

It is important to ensure that the RP's endpoint is not tampered by the adversary. In other OIDC systems, the IdP knows the RP's endpoint during RP registration and thus can verify the endpoint in an identity proof request. However, in UPPRESSO, the IdP sees only a one-time endpoint. So, we let the user verify the correctness of the RP's endpoint using the RP certificate.

3. $\mathbf{PID_{RP}}$ *Registration.* In this phase, the user registers a new RP with PID_{RP} at the IdP using OIDC's dynamic registration. This step has to be conducted by the user but not the RP. Otherwise, the IdP can associate PID_{RP} and ID_{RP}.

3.1 The IdP script sends the PID_{RP} registration request $[PID_{RP}, Hash(N_U), Endpoint_U]$ to the IdP.
3.2 The IdP checks the unexpired PID_{RP}s to verify if the received PID_{RP} is unique. Then, it signs the response as $[PID_{RP}, hash(N_U), validity]_{SK}$, where $validity$ denotes when the PID_{RP} will expire.
3.3 The IdP script forwards the registration result to the RP server through the RP script.
3.4 The RP verifies the IdP's signature, and accepts the result only if PID_{RP} and $hash(N_U)$ match those in the negotiation and PID_{RP} is not expired.

$hash(N_U)$ is a nonce to distinguish different login sessions, because there is a very small chance that a same PID_{RP} is generated for two RPs using different ID_{RP}s and N_Us. This nonce avoids a registration result is acceptable to two RPs.

4. ID Proof Generation. In this phase, the IdP calculates $PID_U = {PID_{RP}}^{ID_U} \bmod p$ and signs the identity proof.

4.1 The RP constructs an identity proof request containing its PID_{RP} and $Endpoint_{RP}$, which is then forwarded to the IdP script through the RP script.
4.2 The IdP authenticates the user if she has not been authenticated yet.

4.3 First, the user checks the scope of the requested attributes, while the IdP script verifies PID_{RP} and $Endpoint_{RP}$ in the request are valid. Then, the IdP script replaces the RP's endpoint with the newly registered one-time $Endpoint_U$ and sends the modified identity proof request to the IdP server.

4.4 The IdP verifies if PID_{RP} and $Endpoint_U$ are registered and unexpired, and then calculates $PID_U = {PID_{RP}}^{ID_U} \bmod p$ for the authenticated user.

4.5 The IdP constructs and signs the identity proof $[PID_{RP}, PID_U, Iss, ValTime, Attr]_{SK}$, where Iss is the identifier of the IdP, $ValTime$ is the validity period, and $Attr$ contains the requested attributes.

4.6 The IdP sends the identity proof to the one-time endpoint. The IdP script forwards the identity proof to the RP script that holds the origin $Endpoint_{RP}$. Finally, the RP script sends it to the RP server.

In this phase, if any check fails, the process will be halted. For example, the user halts the process if PID_{RP} in the identity proof request is inconsistent with the negotiated one. The IdP rejects the identity proof request, if the pair of PID_{RP} and $Endpoint_U$ has not been registered.

5. **Account** *calculation.* The RP verifies the identity proof, derives the user's unique $Account$, and allows her to log in.

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

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

C. Compatibility with OIDC

As described above, UPPRESSO does not introduce any new role nor change the security assumptions for each role. It follows a similar logic flow as OIDC in SSO login and only requires small modifications to perform identifier transformation. Here, we explain the modification in each of the five phases of its SSO login flow to show that UPPRESSO is compatible with OIDC, which indicates UPPRESSO can be easily integrated with other commonly used SSO systems.

Among the five phases, the *scripts downloading* and ***RP*** *identifier transformation* phases are newly introduced by UPPRESSO. The browser is required to download two scripts from the IdP and RP and most of the designed operations in these two phases are performed by the scripts in the browser. So, we requires minimal modifications to the user agent. The other three phases adopt a similar communication pattern as OIDC. In particular, the PID_{RP} *registration* phase can be viewed as a variant of the RP dynamic registration flow of OIDC [28], which allows an entity to register its identity and endpoint at the IdP. Different from OIDC in which only RPs can call a dynamic registration, UPPRESSO allows any authenticated user to launch this process and register an RP identifier with the IdP. The *identity proof generation* and $Account$ *calculation* phases adopt the same steps and functions as the implicit protocol flow of OIDC, while using a few different parameters. First, in identity proof generation, PID_U transformed from ID_U is used to replace ID_U, which is directly supported by OIDC, similar as in the PPID approaches that also convert ID_U into PID_U. The calculation of $Account$ from PID_U can be viewed as a customized step by the RP to derive its user account after the implicit protocol flow of OIDC ends. So,the identity proof generation and $Account$ calculation phases of UPPRESSO can be viewed as a particular but compatible implementation of the implicit protocol flow of OIDC. It is worth noting that the identity proof generation and $Account$ calculation phases of UPPRESSO can be also implemented as the authorization code flow of OIDC with small modifications, which will be discussed in Section VIII.

VI. Security Analysis

We formally analyze the security and privacy properties of UPPRESSO based on the Dolev-Yao style web model [19], which has been widely used in the formal analysis of SSO protocols such as OAuth 2.0 [31] and OIDC [32]. For brevity, we focus on the modifications introduced by UPPRESSO in this paper and neglect the proofs for the security of DNS and HTTPS requests. We refer interested readers to [19] for details.

A. The Web Model

The Dolev-Yao model abstracts the entities in a system, such as browsers and web servers, as *atomic processes*, which communicate with each other through the *events*. [19] also defines *scripting processes* to model client-side scripting such as JavaScript, so a web system consists of a set of atomic and scripting processes. The state of a system, called a *configuration*, consists of the current states of all atomic processes and all the events that can be accepted by these processes. We list the definitions of these notations as below [19].

Messages are defined as formal terms without variables (i.e., ground terms) over a *signature*. The signature Σ consists of a finite set of function symbols (with arity). For messages in this mode, the signature Σ contains constants such as ASCII strings and nonce, sequence symbols such as n-ary sequences $\langle\rangle$, $\langle.\rangle$, $\langle.,.\rangle$, and function symbols that model cryptographic primitives such as `encrypt`, `decrypt` and digital signatures. For example, an HTTP request can be modeled as a ground term containing a type (e.g., `HTTPReq`), a nonce, a method (e.g., `GET` or `POST`), a domain, a path, URL parameters, request headers and a message body, over the Σ in the sequence symbol format. So, an HTTP GET request for the domain exa.com/path?para=1 with empty header and body can be represented as: $m := \langle \mathtt{HTTPReq}, n, \mathtt{GET}, exa.com, /path, \langle\langle para, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$.

Events are the basic communication elements in the model. An event is of the form $\langle a, f, m\rangle$, where a and f represent the addresses of the sender and receiver respectively, and m is the message to be transmitted.

Atomic Processes. An *atomic Dolev-Yao (DY) process* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$, where I^p is the set of addresses that the process listens to, Z^p is the set of states (i.e., terms) that describes the process, s_0^p is an initial state, and R^p is the mapping from an input state $s \in Z^p$ and an event e to a new state s' and an event e'. Each atomic process also contains a set of nonces that it may use.

Scripting Processes represent client-side scripts loaded by the browser to provide server-defined functions to the browser. However, a scripting process must rely on an atomic process, such as the browser, and provide the relation R called by this atomic process.

Equational theory is defined as usual in Dolev-Yao models, which uses the symbol $\equiv$ to represent the congruence relation on terms. For example, $dec(enc(m,k),k) \equiv m$, where k is a symmetric key.

Static equivalence. As defined in [19], two messages t_1 and t_2 are statically equivalent, denoted as $t_1 \approx t_2$, if and only if, for all terms such as $M(x)$ and $N(x)$ which only contain one variable x without nonce, it is true that $M(t_1) \equiv N(t_1)$ **iff** $M(t_2) \equiv N(t_2)$. For instance, for messages m and m', and a symmetric key k, $enc(m,k) \approx enc(m',k)$ is always true to the attacker who does not know k.

Using static equivalence, we define equivalence of $modpow$ function as below. We also define equivalence of HTTP requests and events. Please find the details of the definitions in the Appendix.

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

Web system. We can represent the web infrastructure as a web system of form ($\mathcal{W}$, $\mathcal{S}$, `script`, E^0), where $\mathcal{W}$ is the set of atomic processes containing both honest and malicious processes, $\mathcal{S}$ is the set of scripting processes including honest and malicious scripts, `script` is the set of concrete script codes related to specific scripting processes in $\mathcal{S}$, and E^0 is the set of events acceptable to the processes in $\mathcal{W}$.

A *configuration* of this web system is a tuple (S, E, N), where S is the current states of all processes in $\mathcal{W}$, E is the set of events that the processes accept, and N is a global sequence of nonces that have not been used by the processes yet.

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

B. The Formal Model of UPPRESSO

Accordingly, we model UPPRESSO as a web system, which is defined as $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \texttt{script}, E^0)$. $\mathcal{W}$ is a finite set of atomic processes in UPPRESSO, which contains an IdP server process, a finite set of web servers for the honest RPs, a finite set of honest browsers, and a finite set of attacker processes. Here, we consider all the RP processes and browser processes are honest, and model an RP or a browser controlled by an adversary as an atomic attacker process. $\mathcal{S}$ is a finite set of scripting processes, which contains script_rp, script_idp and script_attacker, where script_rp and script_idp are honest scripts downloaded by an RP process and the IdP process, and script_attacker denotes a script downloaded by an attacker process that exists in all browser processes. Below is a brief description about the processes and scripts in UPPRESSO.

- A browser is an atomic process, which is responsible for sending HTTP requests, receiving HTTP responses, handling user actions, and transmitting messages between scripting processes. As the browsers are considered honest, in the remaining analysis, we focus only on the scripting processes running in the browsers. We refer interested readers to Appendix C and [19] for more details about the browser process.
- The IdP server process (defined as p^i) only accepts the events whose message is an HTTP request with a path in the set of {/script, /dynamicRegistration, /login, /loginInfo, /authorize}. All the events can be accepted by p^i in any state, but the output may vary.
- The RP server process (denoted as p^r) only accepts the events whose message is an HTTP request with a path in {/script, /login, /startNegotiation, /registrationResult, /uploadToken}. However, an event with a path in { /script, /login, /startNegotiation} can be accepted in any state, while an event with a path $\equiv$ /registrationResult is accepted only when the state s is the output of an event whose path $\equiv$ /startNegotiation. Similarly, the following accepted events should have a path in {/registrationResult, /uploadToken}.
- The IdP and RP scripting processes accept the events in the form of HTTP response and postMessage.

C. Security of UPPRESSO

In the security analysis, we consider web systems $\mathcal{UWS}$ defined in Section VI-A. In this model, we considers only one network attacker who is able to listen to and spoof all addresses. The network attacker can control (malicious) browsers and RPs. The analysis of the security of UPPRESSO is to prove the below theorem:

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

In Section III-A, we describe the fundamental security properties that an SSO system should satisfy. Confidentiality and integrity require that the identity proof from the IdP cannot be intercepted or altered. As we assume tall the messages transmitted using HTTPS, we can prove that the encrypted communications over HTTPS between honest entities cannot be tampered by the network attacker. Therefore, an honest RP can receive correct identity proofs and retrieve the correct key for signature verification from the IdP through the honest browsers. For brevity, we do not include the detailed proofs here, which are similar to the proof in [19].

User identification and RP designation informally require that *an attacker should not be able to log in to an honest RP as an honest user*. To prove this property, we assume there exists a UPPRESSO web system in which an attacker can log in to an honest RP as an honest user, and show this assumption leads to a contradiction. We consider the visits to an RP's resource paths are controlled by the visitors' cookie. So, if such a system exists, the attacker could break the security if and only if he owns the cookie bound to the honest user. Based on this, we define a secure UPPRESSO as below.

Definition 2. Let $\mathcal{UWS}$ be a UPPRESSO web system. $\mathcal{UWS}$ is secure *iff* any authenticated cookie $c(u,r)$ of an honest user u for an honest RP $r \in \mathcal{W}$ is unknown to the attacker a.

To prove that an attacker a does not know the authenticated cookie $c(u, r)$, we want to show that (A) a cannot obtain any $c(u, r)$ owned by u; (B) if c is an unauthenticated cookie owned by a, c cannot be set as $c(u, r)$, i.e., being authenticated by r for u; and (C) an honest user u should not use the authenticated cookie of the attacker (i.e., $c(a, r)$). $\mathcal{UWS}$ meeting the requirement (A) can be proved by the following Lemma.

Lemma 1. The cookie owned by an honest user cannot be leaked to the attacker.

First, due to the same-origin policy, an honest browser should not leak the cookie to any attacker. Based on the UPPRESSO model, we also prove that the RP server and the RP script will not send any cookie to other processes. Therefore, the attackers cannot obtain the u's authenticated cookie. Next, to prove $\mathcal{UWS}$ satisfies the requirement (B), we define the process that authenticates a cookie as below.

Definition 3. In $\mathcal{UWS}$, a cookie c is set as an authenticated cookie $c(u, r)$ for a user u and an RP r only when r receives a valid identity proof of u from the owner of c.

Lemma 2. In $\mathcal{UWS}$, an attacker cannot obtain the password of an honest user u.

Lemma 3. In $\mathcal{UWS}$, an attacker cannot forge or modify the proofs issued by the IdP.

Lemma 2 can be easily proved because the password is only sent by an honest IdP scripting process to the IdP server. Lemma 3 can be proved by showing that the proofs issued by the IdP process are signed and verified. With Lemma 2 and Lemma 3, we can prove the following lemma.

Lemma 4. In $\mathcal{UWS}$, an attacker cannot obtain a valid identity proof for an honest user u.

Here, we provide a brief proof for Lemma 4. A valid identity proof can only be obtained from one of the four processes: the IdP server process, the RP server process, the IdP scripting process and the RP scripting process. According to the model, the honest RP scripting processes only send identity proofs to an honest RP server, while the RP server never sends the proofs to any other process. So, only the process that holds u's password can obtain u's identity proof from the IdP server. As the attacker does not know u's password, he cannot receive the identity proof of u from the IdP server process. Finally, it is a little complicated to prove that the attacker cannot obtain the identity proof from the IdP scripting process. So, we only describe it intuitively. That is, an honest user u only sends the identity proof from the IdP scripting process to the receiver specified by the RP certificate $cert_r$. And, an identity proof is valid to an honest RP r only if $cert_r$ belongs to r (we include a full proof in the Appendix).

Next, we prove a $\mathcal{UWS}$ system meets the requirement (C). First, the attacker cannot set $c(a, r)$ with the RP's origin in an honest browser due to the same-origin policy. According to Definition 3, the RP r sets $c(a, r)$ in an honest browser u if it receives an identity proof with the attacker's PID_U and a valid PID_{RP} generated by u and r. This requires the attacker to know a valid PID_{RP}. According to the following lemma, the attacker cannot obtain an identity proof with a valid PID_{RP}.

Lemma 5. The attacker cannot know a valid PID_{RP} negotiated by a user u and an RP r.

Finally, we prove $\mathcal{UWS}$ satisfies requirements (A), (B) and (C) in Definition 2. As a result, Theorem 1 is proved. Due to space limit, we include all the detailed proofs of the lemmas and theorems in the Appendix.

D. Privacy of UPPRESSO

We adopt a similar web model as above, which contains web attackers instead of network attackers. Then, we show the privacy of UPPRESSO by proving Theorem 2.

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

First, we define IdP-Privacy and RP-Privacy as follows.

Definition 4. IdP-Privacy. Let $\mathcal{UWS}$ be a UPPRESSO web system. Given honest RPs $r_1, r_2 \in \mathcal{W}$, the IdP $i \in \mathcal{W}$ and an honest user u, $\mathcal{UWS}$ is IdP-Privacy *iff* for each event e_1 received by i associate with to u's login session to r_1, there always exists an event e_2 associated with u's login session to r_2, and e_1 and e_2 are equivalent.

Here, we provide a brief proof that $\mathcal{UWS}$ meets the requirements defined in Definition 4. First, all events to the IdP over HTTPS should be considered as equivalent to the web attacker. Since the IdP server is assumed honest but curious, i holds only the events to an IdP server process and does not attempt to fetch parameters from other processes or set illegal parameters in the system. Let us consider multiple requests from a same user. The IdP server only accepts the events whose message is an HTTP request with a $path$ in the set of {/script, /dynamicRegistration, /login, /loginInfo, /authorize} that are visited in each login session. Since the visits to /script and /loginInfo carry no parameter nor body, these events from two different login sessions are considered equivalent to i. Moreover, as the visits to /login only carry u's username and password, these events are also equivalent. Finally, the visits to /dynamicRegistration and /authorize carry PID_{RP}s and $endpoint$s, where PID_{RP}s are statically equivalent because of Definition 1 and $endpoint$s are unrelated random constants, thus these events are also considered equivalent.

Definition 5. RP-Privacy Let $\mathcal{UWS}$ be a UPPRESSO web system. For honest RPs $r_1, r_2 \in \mathcal{W}$ and honest users u_1 and u_2, $\mathcal{UWS}$ is RP-Privacy *iff* r_1 and r_2 share states,

- for each event e_1 received by r_2 that is associated with u_1's login session to r_2, there always exists an event e_2 associated with u_2's login in to r_2 and e_1 and e_2 are equivalent to r_1.
- for each event received by r_2, the event cannot be directly linked to an existing user's attributes at r_1.

The RP server process accepts the events whose message is an HTTP request with a $path$ in {/script, /login, /startNegotiation, /registrationResult, /uploadToken}. When the RP is malicious, we should consider the events received by the RP scripting process. However, as all the messages received by the RP scripting process are transmitted to the RP server, we focus only on the events received by the RP server.

Similar, we can prove $\mathcal{UWS}$ meets the requirements defined in Definition 5. First, we assume all the parameters are set legally. As the events visiting /script and /login carry no parameter and body, they are equivalent. Since the visits to /startNegotiation carry only the nonce, these events are also equivalent. The visits to /registrationResult carry the registration result, which contains PID_{RP}, N_U and $endpint$ and is signed by the IdP. As the content of the result can be viewed as random constant, the events can also be considered as equivalent. The visits to /uploadToken includes the identity proof containing PID_{RP} and PID_U. According to Definition 1, PID_Us are statically equivalent to r_1. Finally, even when r_2 share state with r_1, r_1 still cannot convert an $Account_{r_1}$ to an account $Account_{r_2}$ at r_2, so that the events cannot be linked to an existing user.

Then, we consider the case that malicious RPs exist in the system. According to Definition 1, PID_U and $Accounts$ seem equivalent to the attacker who does not know ID_U. However, the attacker cannot obtain ID_U from other UPPRESSO processes, since the IdP never sends out ID_U in clear. Next, we prove that a malicious RP cannot derive ID_U from PID_U or $Account$, even if it can manipulate the generation of PID_U and $Account$. First, the malicious RP may use a forged ID_{RP} to make PID_U or $Account$ inequivalent. However, such illegal ID_{RP} can be detected by the IdP scripting process of an honest user. Moreover, the malicious RP may make a same user submit the identity proof with the same PID_U. However, as PID_U is generated with a nonce N_U chosen by the user, the RP cannot manipulate PID_U. Now, we prove that system meets all the requirements in Definition 5. Therefore, Theorem 2 is proved.

VII. Implementation and Performance Evaluation

We have implemented the UPPRESSO prototype, and evaluated its performance by comparing with the original OIDC which only prevents RP-based identity linkage, and SPRESSO which only prevents IdP-based login tracing.

A. Implementation

We adopt SHA-256 for digest generation, and RSA-2048 for signature generation. We randomly choose a 2048-bit prime as p, a 256-bit prime as q, and the q-order generators as ID_{RP}. N_U and ID_U are 256-bit random numbers. Then, the discrete logarithm problem provides equivalent security strength (i.e., 112 bits) as RSA-2048 [43]. UPPRESSO includes the processing at the IdP, users and the RPs. The implementations at each entity are as follows.

The implementation of the IdP only needs small modifications on the existing OIDC implementation. The UPPRESSO IdP is implemented based on MITREid Connect [44], an open-source OIDC Java implementation certificated by the OpenID Foundation [45]. We add 3 lines of Java code to calculate PID_U, about 20 lines to modify the way to send identity proof to the RP, about 50 lines to the function of dynamic registration to support PID_{RP} registration, i.e., checking PID_{RP} and adding a signature and validity period in the response. The calculations of ID_{RP}, PID_U and RSA signature are implemented based on Java built-in cryptographic libraries (e.g., BigInteger).

The user-side processing is implemented as a JavaScript code provided by IdP and RP server, respectively containing about 200 lines and 150 lines of codes, to provide the functions in Steps 2.1, 2.3 and 4.3. The cryptographic computations, e.g., $Cert_{RP}$ verification and PID_{RP} negotiation, are implemented based on jsrsasign [46], an efficient JavaScript cryptographic library.

We provide a Java SDK for RPs to integrate UPPRESSO. The SDK provides 2 functions to encapsulate RP's processings: one for RP identifier transformation, PID_{RP} registration and identity proof request generation; while the other for identity proof verification and $Account$ calculation. The SDK is implemented based on the Spring Boot framework with about 1000 lines code, and cryptographic computations are implemented based on Spring Security library. An RP only needs to invoke these two functions for the integration.

B. Performance Evaluation

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

Setting. We compare UPPRESSO with MITREid Connect [44] and SPRESSO [19], where MITREid Connect provides open-source Java implementations [44] of IdP and RP's SDK, and SPRESSO provides the JavaScript implementations based on node.js for all entities [19]. We implemented a Java RP based on Spring Boot framework for UPPRESSO and MITREid Connect, by integrating the corresponding SDK respectively. The RPs in all the three schemes provide the same function, i.e., extracting the user's account from the identity proof. We have measured the time for a user's login at an RP, and calculated the average values of 1000 measurements. For better analysis, we divide a login into 4 phases according to the lifecycle of identity proof: **Identity proof requesting** (Steps 1.1-4.3 in Figure 3), the RP (and user) constructing and transmitting the request to IdP; **Identity proof generation** (Steps 4.4 and 4.5 in Figure 3), the IdP generating identity proof (no user authentication); **Identity proof extraction** (Steps 4.5 and 4.6 in Figure 3), the RP server extracts the identity proof from the IdP; and **Identity proof verification** (Steps 5.1 and 5.2 in Figure 3), the RP verifying and parsing the identity proof.

Results. The evaluation results are provided in Figure 4. The overall processing times are 113 ms, 308 ms and 492 ms for MITREid Connect, SPRESSO and UPPRESSO, respectively. The main overhead in UPPRESSO is opening the new window and downloading the script from IdP, which needs about 255 ms. This overhead could be reduced by implicitly conducting this procedure when the user visits the RP website. The details are as follows.

In the requesting, UPPRESSO requires that (1) the user downloads the RP script, opens the IdP window and downloads the IdP script, and (2) the user and RP performs 1 modular exponentiations for RP identifier transformation and complete PID_{RP} registration at the IdP. The total proceeding time is 371 ms (255 ms downloading scripts and open new window

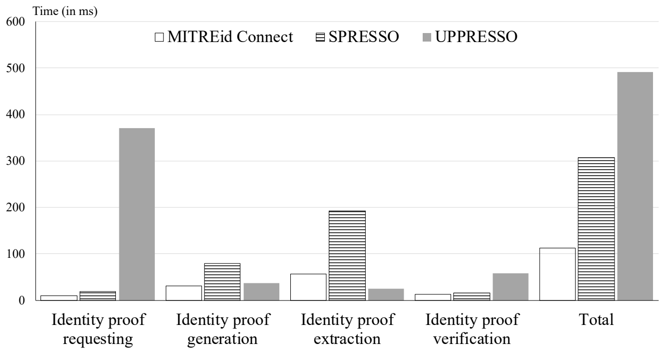

Fig. 4: The Evaluation.

), where SPRESSO needs 19 ms for the RP to obtain IdP's public key and encrypt its domain; while MITREid Connect only needs 10 ms.

In the generation, UPPRESSO needs an extra 6 ms for computing PID_U, compared to MITREid Connect which only needs 32 ms. SPRESSO requires 71 ms, as it implements the IdP based on node.js and therefore can only adopt a JavaScript cryptographic library, while others adopt a more efficient Java library. As the processings in SPRESSO and MITREid Connect are the same, the processing time in SPRESSO may be reduced to 32 ms.

In the identity proof extraction, UPPRESSO only needs about 25 ms where the scripts relay the identity proof to the RP server. MITREid Connect requires the IdP to send the identity proof to the RP's web page which then sends the proof to the RP server through a JavaScript function, and needs 57 ms. SPRESSO needs the longest time (193 ms) due to a complicated processing at the user's browser, which needs the browser to obtain identity proofs from the IdP, download the JavaScript program from a trusted entity (forwarder), execute the program to decrypt RP's endpoint, send identity proofs to this endpoint (an RP's web page) who finally transmits the proof to RP server. In the evaluation, the forwarder and IdP are deployed in one machine, which doesn't introduce performance degradation based on the observation.

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

VIII. Discussions and Future work

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

Scalability. The adversary cannot exhaust ID_{RP} and PID_{RP}. For ID_{RP}, it is generated only in RP's initial registration. For PID_{RP}, in practice, we only need to ensure all PID_{RP}s are different among the unexpired identity proof (the number denoted as n). We assume that IdP doesn't perform the uniqueness check, and then calculate the probability that at least two PID_{RP}s are equal in these n ones. The probability is $1-\prod_{i=0}^{n-1}(1-i/q)$ which increases with n. For an IdP with throughput $2*10^8$ req/s, when the validity period of the identity proof (PID_{RP}) is set as 5 minutes, n is less than 2^{36}, then the probability is less than 2^{-183} for 256-bit q. Moreover, as this probability is negligible, the uniqueness check of PID_{RP}, i.e., the PID_{RP} registration, could be removed in the SSO login process, and this optimization can be adopted when this negligible probability is acceptable by the users and RPs.

Security against DoS attack. The adversary may attempt to perform DoS attack on the IdP and RP. For example, the adversary may act as a user to invoke the PID_{RP} registration (Step 3.1) and identity proof generation (Step 4.3) at the IdP, which requires the IdP to perform two signature generations and one modular exponentiation. However, as the user has already been authenticated at the IdP, the IdP could identify the malicious users based on audit, in addition to the existing DoS mitigation schemes.

OIDC authorization code flow support. The privacy-preserving functions $\mathcal{F}_{ID_U \mapsto PID_U}$, $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ and $\mathcal{F}_{PID_U \mapsto Account}$ can be integrated into OIDC authorization code flow directly, therefore RP-based identity linkage and IdP-based login tracing are still prevented during the construction and parsing of identity proof. The only privacy leakage is introduced by the transmission, as RP servers obtain the identity proof directly from the IdP in this flow, which allows the IdP to obtain RP's network information (e.g., IP address). UPPRESSO needs to integrate existing anonymous networks (e.g., Tor) to prevent this leakage.

Malicious IdP mitigation. The IdP is assumed to assign a unique ID_{RP} in $Cert_{RP}$ for each RP and generate the correct PID_U for each login. The malicious IdP may attempt to provide the incorrect ID_{RP} and PID_U, which could be prevented by integrating certificate transparency [47] and user's identifier check [19]. With certificate transparency [47], the monitors check the uniqueness of ID_{RP} among all the certificates stored in the log server. To prevent the malicious IdP from injecting any incorrect PID_U, the user could provide a nickname to the RP for an extra check as in SPRESSO [19].

Identity linkage through cookie. In UPPRESSO, IdP does not provide any distinctive information (such as ID_U) of each user to RP, which avoids RP-based identity linkage. However, the cookie of each user may be exploited by RPs to correlate the same user. For instance, while the user has logged in to RP_A with $Account_A$, RP may redirect user's $Account_A$ to RP_B through the hidden iframe. That is, as long as the user has logged in to RP_B with $Account_B$, the user would be correlated. Moreover, this attack is not only appeared in SSO systems, but also existed in all user-account systems. However, this attack can be easily detected through multiple methods, such as checking the iframe in script, observing redirection flow through browser network tool, and detecting the redirection based on the browser extension.

IX. Related Works

Various SSO protocols have been proposed, such as, OIDC, OAuth 2.0, SAML, Central Authentication Service (CAS) [48] and Kerberos [49]. These protocols are widely adopted in Google, Facebook, Shibboleth project [50], Java applications and etc. And, plenty of works have been conducted on privacy protection and security analysis for SSO systems.

A. Privacy protection for SSO systems.

Privacy-preserving SSO systems. As suggested by NIST [17], SSO systems should prevent both RP-based identity linkage and IdP-based login tracing. The pairwise user identifier is adopted in SAML [3] and OIDC [1], and only prevents RP–based identity linkage; while SPRESSO [19] and BrowserID [18] only prevent IdP-based login tracing. BrowserID is adopted in Persona [27] and Firefox Accounts [21], however an analysis on Persona found IdP-based login tracing could still succeed [18], [51]. UPPRESSO prevents both the RP-based identity linkage and IdP-based login tracing, and could be integrated into OIDC which has been formally analyzed [32].

Anonymous SSO systems. Anonymous SSO schemes are designed to allow users to access a service (i.e. RP) protected by a verifier (i.e., IdP) without revealing their identities. One of the earliest anonymous SSO systems was proposed for Global System for Mobile (GSM) communication in 2008 [52]. The notion of anonymous SSO was formalized [53] in 2013. And, various cryptographic primitives, such as group signature, zero-knowledge proof and etc., were adopted to design anonymous SSO schemes [53], [54]. Anonymous SSO schemes are designed for the anonymous services, and not applicable to common services which need user identification.

B. Security analysis of SSO systems.

Formal analysis on SSO standards. The SSO standards (e.g., SAML, OAuth and OIDC) have been formally analyzed. Fett et al. [31], [32] have conducted the formal analysis on OAuth 2.0 and OIDC standards based on an expressive Dolev-Yao style model [51], and proposed two new attacks, i.e., 307 redirect attack and IdP Mix-Up attack. When the IdP misuses HTTP 307 status code for redirection, the sensitive information (e.g., credentials) entered at the IdP will be leaked to the RP by the user's browser. While, IdP Mix-Up attack confuses the RP about which IdP is used and makes the victim RP send the identity proof to the malicious IdP, which breaks the confidentiality of the identity proof. Fett et al. [31], [32] have proved that OAuth 2.0 and OIDC are secure once these two attacks prevented. UPPRESSO could be integrated into OIDC, which simplifies its security analysis. [30] formally analyzed SAML and its variant proposed by Google, and found that Google's variant of SAML doesn't set RP's identifier in the identity proof, which breaks RP designation.

Single sign-off. In SSO systems, once a user's IdP account is compromised, the adversary could hijack all her RPs' accounts. A backwards-compatible extension, named single sign-off, is proposed for OIDC. The single sign-off allows the user to revoke all her identity proofs and notify all RPs to freeze her accounts [5]. The single sign-off could also be achieved in UPPRESSO, where the user needs to revoke the identity proofs at all RPs, as the IdP doesn't know which RPs the user visits.

Analysis on SSO implementations. Various vulnerabilities were found in SSO implementations, and then exploited for impersonation and identity injection attacks by breaking the confidentiality [6], [7], [9]–[11], integrity [6], [8], [11]–[14] or RP designation [11]–[15] of identity proof. Wang et al. [6] analyzed the SSO implementations of Google and Facebook from the view of the browser relayed traffic, and found logic flaws in IdPs and RPs to break the confidentiality and integrity of identity proof. An authentication flaw was found in Google Apps [9], allowing a malicious RP to hijack a user's authentication attempt and inject the malicious code to steal the cookie (or identity proof) for the targeted RP, breaking the confidentiality. The integrity has been tampered with in SAML, OAuth and OIDC systems [6], [8], [12]–[14], due to various vulnerabilities, such as XML Signature wrapping (XSW) [8], RP's incomplete verification [6], [12], [14], IdP spoofing [13], [14] and etc. And, a dedicated, bidirectional authenticated secure channel was proposed to improve the confidentiality and integrity of identity proof [39]. The vulnerabilities were also found to break the RP designation, such as the incorrect binding at IdPs [12], [15], insufficient verification at RPs [13]–[15]. Automatical tools, such as SSOScan [33], OAuthTester [35] and S3KVetter [15], have been designed to detect vulnerabilities for breaking the confidentiality, integrity or RP designation of identity proof.

Analysis on mobile SSO systems. In mobile SSO systems, the IdP App, IdP-provided SDK (e.g., an encapsulated WebView) or system browser are adopted to redirect identity proof from IdP App to RP App. However, none of them was trusted to ensure that the identity proof could be only sent to the designated RP [34], [38], as WebView and system browser cannot authenticate RP App while the IdP App may be repackaged. Moreover, the SSO protocols needed to be modified to provide SSO services for mobile Apps, however these modifications were not well understood by RP developers [36], [38]. The top Android applications have been analyzed [34], [36]–[38], [40], and vulnerabilities were found to break the confidentiality [34], [36]–[38], [40], integrity [36], [38], and RP designation [37], [38] of identity proof.

X. CONCLUSION

In this paper, we propose UPPRESSO, an unlinkable privacy-preserving single sign-on system, which protects a user's login activities at different RPs against both curious IdP and collusive RPs. To the best of our knowledge, UPPRESSO is the first approach that defend against both IdP-based login tracing and RP-based identity linkage privacy threats at the same time. To achieve these goals, we convert the privacy problem in SSO services into an identifier-transformation problem and design three transformation functions based on the discrete logarithm problem, where $\mathcal{F}_{ID_{RP} \mapsto PID_{RP}}$ prevents curious IdP from knowing the identity of the RP, $\mathcal{F}_{ID_U \mapsto PID_U}$ prevents collusive RPs from linking a user based on her identifier, and $\mathcal{F}_{PID_U \mapsto Account}$ allows each RP to derive an identical account for a user in her multiple logins. The three functions could be integrated with existing SSO protocols, such as OIDC, to enhance the protection of user privacy, without breaking any security guarantee of SSO. Moreover, the evaluation on the prototype of UPPRESSO demonstrates that it supports an efficient SSO service, where a single login takes only 492 ms on average.

REFERENCES

[1] 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.

[2] Dick Hardt, "The OAuth 2.0 Authorization Framework," *RFC*, vol. 6749, pp. 1–76, 2012.

[3] 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.

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

[5] 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 *Proceedings of the 27th USENIX Security Symposium, USENIX Security, Baltimore, MD, USA*, 2018, pp. 1475–1492.

[6] 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 *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), San Francisco, California, USA*, 2012, pp. 365–379.

[7] San-Tsai Sun and Konstantin Beznosov, "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems," in *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security (CCS), Raleigh, NC, USA*, 2012, pp. 378–390.

[8] 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.

[9] 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.

[10] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis, "Discovering concrete attacks on website authorization by formal analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

[11] Wanpeng Li and Chris J. Mitchell, "Analysing the security of Google's implementation of OpenID Connect," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), San Sebastián, Spain*, 2016, pp. 357–376.

[12] 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), Los Angeles, CA, USA*, 2016, pp. 167–176.

[13] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk, "Do not trust me: Using malicious IdPs for analyzing and attacking single sign-on," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrücken, Germany*, 2016, pp. 321–336.

[14] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich, "Sok: Single sign-on security - an evaluation of OpenID Connect," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France*, 2017, pp. 251–266.

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

[16] Eve Maler and Drummond Reed, "The venn of identity: Options and issues in federated identity management," *IEEE Security & Privacy*, vol. 6, no. 2, pp. 16–23, 2008.

[17] Paul A Grassi, M Garcia, and J Fenton, "NIST special publication 800-63c digital identity guidelines: Federation and assertions," *National Institute of Standards and Technology, Los Altos, CA*, 2017.

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

[19] 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 (CCS), Denver, CO, USA*, 2015, pp. 1358–1369.

[20] "Google identity platform," https://developers.google.com/identity/, Accessed August 20, 2019.

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

[22] 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.

[23] 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.

[24] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant, "Information sharing across private databases," in *Proceedings of the 30th ACM International Conference on Management of Data (SIGMOD), San Diego, California, USA*, 2003, pp. 86–97.

[25] Emiliano De Cristofaro and Gene Tsudik, "Practical private set intersection protocols with linear complexity," in *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC), Canary Islands, Spain*, 2010, pp. 143–159.

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

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

[28] 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.

[29] Taher El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of 4th International Cryptology Conference (CRYPTO), Santa Barbara, California, USA*, 1984, pp. 10–18.

[30] 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), Alexandria, VA, USA*, 2008, pp. 1–10.

[31] 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 (CCS), Vienna, Austria*, 2016, pp. 1204–1215.

[32] Daniel Fett, Ralf Küsters, and Guido Schmitz, "The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines," in *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF), Santa Barbara, CA, USA*, 2017, pp. 189–202.

[33] 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.

[34] 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 (ACSAC), Los Angeles, CA, USA*, 2015, pp. 61–70.

[35] 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), Xi'an, China*, 2016, pp. 651–662.

[36] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi, "Breaking and fixing mobile app authentication with OAuth2.0-based protocols," in *Proceedings of the 15th International Conference on Applied Cryptography and Network Security (ACNS), Kanazawa, Japan*, 2017, pp. 313–335.

[37] 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 14th ACM Asia Conference on Computer and Communications Security (AsiaCCS), Auckland, New Zealand*, 2019, pp. 269–282.

[38] 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 (CCS), Scottsdale, AZ, USA, 2014, pp. 892–903.

[39] 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 *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Gothenburg, Sweden.*, 2014, pp. 276–298.

[40] Mohamed Shehab and Fadi Mohsen, "Towards enhancing the security of OAuth implementations in smart phones," in *Proceedings of the 3rd IEEE International Conference on Mobile Services, Anchorage, AK, USA*, 2014, pp. 39–46.

[41] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang, "Targeted online password guessing: An underestimated threat," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria*, 2016, pp. 1242–1254.

[42] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin, "oPass: A user authentication protocol resistant to password stealing and password reuse attacks," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 7, no. 2, pp. 651–663, 2012.

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

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

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

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

[47] Ben Laurie, Adam Langley, and Emilia Käsper, "Certificate transparency," *RFC*, vol. 6962, pp. 1–27, 2013.

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

[49] 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.

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

[51] Daniel Fett, Ralf Küsters, and Guido Schmitz, "An expressive model for the web infrastructure: Definition and application to the BrowserID SSO system," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P), Berkeley, CA, USA*, 2014, pp. 673–688.

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

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

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

Appendix A
Web Model

A. Data Format

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

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

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

An HTTP response message is the term of the form

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

The details are defined as follows:

- `HTTPReq` and `HTTPResp` denote the types of messages.
- $nonce$ is a random number that maps the response to the corresponding request.
- $method$ is one of the HTTP methods, such as `GET` and `POST`.
- $host$ is the constant string domain of visited server.
- $path$ is the constant string representing the concrete resource of the server.
- $parameters$ contains the parameters carried by the url as the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, for example, the $parameters$ in the url $http://www.example.com?type = confirm$ is $\langle\langle type, confirm\rangle\rangle$.
- $headers$ is the header content of each HTTP messages as the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, such as $\langle\langle Referer, http://www.example.com\rangle, \langle Cookies, c\rangle\rangle$.
- $body$ is the body content carried by HTTP `POST` request or HTTP response in the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$.
- $status$ is the HTTP status code defined by HTTP standard.

URL. URL is a term $\langle \texttt{URL}, protocol, host, path, parameters\rangle$, where `URL` is the type, $protocol$ is chosen in $\{\texttt{S}, \texttt{P}\}$ as `S` stands for HTTPS and `P` stands for HTTP. The $host, path$, and $parameters$ are the same as in HTTP messages.

Origin. An Origin is a term $\langle host, protocol\rangle$ that stands for the specific domain used by the HTTP CORS policy, where $host$ and $protocol$ are the same as in URL.

POSTMESSAGE. PostMessage is used in the browser for transmitting messages between scripts from different origins. We define the postMessage as the form $\langle \texttt{POSTMESSAGE}, target, Content, Origin\rangle$, where `POSTMESSAGE` is the type, $target$ is the constant nonce which stands for the receiver, $Content$ is the message transmitted and $Origin$ restricts the receiver's origin.

XMLHTTPREQUEST. XMLHTTPRequest is the HTTP message transmitted by scripts in the browser. That is, the XMLHTTPRequest is converted from the HTTP message by the browser. The XMLHTTPRequest in the form $\langle \texttt{XMLHTTPREQUEST}, URL, methods, Body, nonce\rangle$ can be converted into HTTP request message by the browser, and $\langle \texttt{XMLHTTPREQUEST}, Body, nonce\rangle$ is converted from HTTP response message.

Data Operation. The data used in UPPRESSO are defined in the following forms:

- **Standardized Data** is the data in the fixed format, for instance, the HTTP request is the standardized data in the form $\langle$`HTTPReq`, $nonce$, $method$, $host$, $path$, $parameters$, $headers$, $body\rangle$. We assume there is an HTTP request $r := \langle$`HTTPReq`, n, `GET`, $example.com$, $/path$, $\langle\rangle$, $\langle\rangle$, $\langle\rangle\rangle$, here we define the operation on the r. That is, the elements in r can be accessed in the form $r.name$, such that $r.method \equiv$ `GET`, $r.path \equiv /path$ and $r.body \equiv \langle\rangle$.
- **Dictionary Data** is the data in the form $\langle\langle name, value\rangle, \langle name, value\rangle, \ldots\rangle$, for instance the $body$ in HTTP request is dictionary data. We assume there is a $body := \langle\langle username, alice\rangle, \langle password, 123\rangle\rangle$, here we define the operation on the $body$. That is, we can access the elements in $body$ in the form $body[name]$, such that $body[username] \equiv alice$ and $body[password] \equiv 123$. We can also add the new attributes to the dictionary, for example after we set $body[age] := 18$, the $body$ are changed into$\langle\langle username, alice\rangle, \langle password, 123\rangle, \langle age, 18\rangle\rangle$.

Patten Matching. We define the term with the variable $*$ as the pattern, such as $\langle a, b, *\rangle$. The pattern matches any term which only replaces the $*$ with other terms. For instance, $\langle a, b, *\rangle$ matches $\langle a, b, c\rangle$.

B. Browser Model

In UPPRESSO, we assume that the browsers are honest, therefore, we only need to analyze how the browsers interactive with the scripts.

We firstly introduce the windows and documents of the browser model.

Window. A window w is a term of the form $w = \langle nonce, documents, opener\rangle$, representing the the concrete browser window in the system. The $nonce$ is the window reference to identify each windows. The $documents$ is the set of documents (defined below) including the current document and cached documents (for example, the documents can be viewed via the "forward" and "back" buttons in the browser). The $opener$ represents the window in which this window is created, for instance, while a user clicks the href in document d and it creates a new window w, there is $w.opener \equiv d.nonce$.

Document. A document d is a term of the form

$$\langle nonce, location, referrer, script, scriptstate, \\ scriptinputs, subwindows, active\rangle$$

where document is the HTML content in the window. The $nonce$ locates the document. $Location$ is the URL where the document is loaded. $Referrer$ is same as the Referer header defined in HTTP standard. The $script$ is the scripting process downloaded from each servers. $scriptstate$ is define by the script, different in each scripts. The $scriptinputs$ is the message transmitted into the scripting process. The $subwindows$ is the set of $nonce$ of document's created windows. $active$ represents whether this document is active or not.

A scripting process is the dependent process relying on the browser, which can be considered as a relation R mapping a message input and a message output. And finally the browser will conduct the command in the output message. Here we give the description of the form of input and output.

- **Scripting Message Input.** The input is the term in the form

$$\langle tree, docnonce, scriptstate, stateinputs, cookies, \\ localStorage, sessionStorage, ids, secret\rangle$$

- **Scripting Message Output.** The output is the term in the form

$$\langle scriptstate, cookies, localStorage, \\ sessionStorage, command\rangle$$

The $tree$ is the relations of the opened windows and documents, which are visible to this script. $Docnonce$ is the document nonce. The $Scriptstate$ is a term of the form defined by each script. $Scriptinputs$ is the message transmitted to script. However, the $scriptinputs$ is defined as standardized forms, for example, postMessage is one of the forms of $scriptinputs$. $Cookies$ is the set of cookies that belong to the document's origin. $LocalStorage$ is the storage space for browser and $sessionStorage$ is the space for each HTTP sessions. Ids is the set of user IDs while $secret$ is the password to corresponding user ID. The $command$ is the operation which is to be conducted by the browser. Here we only introduce the form of commands used in UPPRESSO system. We have defined the postMessage and XMLHTTPRequest (for HTTP request) message which are the $commands$. Moreover, a term in the form $\langle \texttt{IFRAME}, URL, WindowNonce\rangle$ asks the browser to create this document's subwindow and it visits the server with the URL.

C. Model of UPPRESSO

In this section, we introduce the model of processes in UPPRESSO system, including IdP server process, RP server process, IdP scripting process and RP scripting process. We will focus on the state form and relation R. They can describe that what kind of event can be accepted by the process in each state, and the content of new output events and states.

D. IdP Server Process

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

- ID is the identifier of IdP.
- p is the large prime mentioned before.
- $SignKey$ is the private key used by IdP to generate signatures.
- $sessions$ is the term in the form of $\langle\langle Cookie, session\rangle\rangle$, the Cookie uniquely identifies

the session and sessions store the browser uploaded messages.

- $users$ is the set of user's information, including $username, password, ID_U$ and other attributes.
- RPs is the set of RP information which consists of ID of RP (PID_{RP}), $Endpoints$ (i.e., the set of RP's validity endpoints) and $Validity$.
- $Validity$ is the validity for IdP generated signatures.
- $Tokens$ is the set of IdP generated Identity proofs.

To make the description clearer, we also provide the $functions$ to define the complicated procedure.

- $\texttt{SecretOfID(u)}$ is used to search the user u's password.
- $\texttt{UIDOfUser(u)}$ is used to search the user u's ID_U.
- $\texttt{ListOfPID}()$ is the set of IDs of registered RP.
- $\texttt{EndpointsOfRP(r)}$ is the set of endpoints registered by the RP with ID r.
- $\texttt{ModPow(a, b, c)}$ is the result of $a^b \mod c$.
- $\texttt{CurrentTime}()$ is the system current time.

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

E. RP process

The state of RP server process is a term in the form $\langle ID_{RP}, Endpoints, IdP, Cert, sessions, users\rangle$. Other attributes are not mentioned here.

- ID_{RP} and $Endpoints$ are RP's registered information at IdP.
- $Cert$ is the IdP signed RP information containing $ID_{RP}, Endpoints$ and other attributes.
- IdP is the term of the form $\langle ScriptUrl$, p, q, $PubKey\rangle$, where $ScriptUrl$ is the site to download IdP script, p and q are large primes defined before, and $PubKey$ is the public key used to verify the IdP signed messages.
- $sessions$ is same as it in IdP process.
- $users$ is the set of users registered at this RP, each user is uniquely identified by the $Account$.

The new $functions$ are defined as follows

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

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

F. IdP scripting process

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

- $IdPDomain$ is the IdP's host.
- $Parameters$ is used to store the parameters received from other processes.
- p is the large prime defined before.
- q is used to label the procedure point in the login.
- $refXHR$ is the nonce to map HTTP request and response.

The new $functions$ are defined as follows.

- $\texttt{PARENTWINDOW}(tree, docnonce)$. The first parameter is the input relation tree defined before, and the second parameter is the nonce of a document. The output returned by the function is the current window's opener's nonce (null if it doesn't exist nor it is invisible to this document).
- $\texttt{CHOOSEINPUT}(inputs, pattern)$. The first parameter is a set of messages, and the second parameter is a pattern. The result returned by the function is the message in $inputs$ matching the $pattern$.
- $\texttt{RandomUrl}()$ returns a newly generated host string.

The relation of IdP scripting process $script\_idp$ is shown in Appendix B Algorithm 3.

G. RP scripting process

The state of RP scripting process $scriptstate$ is a term in the form $\langle IdPDomain$, $RPDomain$, $Parameters$, q, $refXHR\rangle$. The $RPDomain$ is the host string of the corresponding RP server, and other terms are defined in the same way as in IdP scripting process.

Here, we define the function $\texttt{SUBWINDOW}(tree, docnonce)$, which takes the $tree$ defined above and the current document's $nonce$ as the input. And it selects the $nonce$ of the first window opened by this document as the output. However, if there is no opened windows, it returns null.

The relation of RP scripting process $script\_rp$ is shown in Appendix B Algorithm 4.

H. Proof of Theorem 1

We assume that all the network messages are transmitted using HTTPS, postMessage messages are protected by the browser, and the browsers are honest, so web attackers can never break the security of UPPRESSO.

We provide the detailed proof on the security of UPPRESSO. As analyzed above, the security requirements of UPPRESSO are that the system must ensure only the legitimate user can log into an honest RP under her unique account. We consider the visits to RP's resource paths are controlled by the visitors' cookies, so that the attacker can break the security only when he owns the cookie bound to the honest user. Therefore, we can propose the Definition 2 about the secure UPPRESSO system.

Definition 1. Let $\mathcal{UWS}$ be a UPPRESSO web system, $\mathcal{UWS}$ is secure **iff** for any honest RP $r \in \mathcal{W}$ and the authenticated cookie c for honest u, c is unknown to the attacker a and only c is used by u.

Therefore, the proof of Theorem 1 is converted into whether the UPPRESSO system meets the requirement in Definition 2. However, as we consider the attacker initially does not know any honest user's cookie, the requirement of Definition 2 can be separated as the following requirements. Before describing the requirements, we firstly define the user u's authenticated cookie for RP r as $c(u, r)$.

Requirement 1. If $c(u, r)$ is the authenticated cookie owned by u, $c(u, r)$ cannot be obtained by a.

Requirement 2. If c is an unauthenticated cookie owned by a, c cannot be set as $c(u, r)$.

Requirement 3. The user u does not use the attacker's cookie (denoted as $c(a, r)$).

To prove that UPPRESSO meets the requirements, we now provide the following lemmas. Lemma 1 proves that the UPPRESSO system meets Requirement 1.

Lemma 6. Attacker does not learn users' cookies.

Proof: The Brute-force attacks, such as exhausting the possible users' cookies, are infeasible due to the length of cookies. The attackers can only try to obtain the cookies from honest processes in the system. For an honest user u and the honest RP r, the valid cookie $c(u, r)$ can only be obtained by u's browser b_u, the r's script $script\_rp$ and RP's server P^r . Here, we only need to prove that the attacker cannot receive the event from these processes which carries $c(u, r)$.

- b. The browsers are considered honest and well implemented. Therefore, based on the same-origin policy, b_u only sends r's cookie to RP's domain, so that attackers cannot receive the cookie.
- $script\_rp$. According to Algorithm 4, the $script\_rp$ does not send any cookie.
- P^r. According to Algorithm 2, the P^r does not send any cookie.

Therefore, Lemma 6 is proved. ■

To prove UPPRESSO system meets Requirement 2, we need to know how the cookie can be set as $c(u, r)$. Based on Algorithm 2, we propose the following definition.

Definition 2. In $\mathcal{UWS}$, the cookie c is to be set as $c(u, r)$ only when RP r receives a valid identity proof (denoted as $t(u, r)$ here) of u, from the owner of c.

To prove that $t(u, r)$ cannot be obtained by attackers, we introduce the following lemmas.

Lemma 1. Attacker does not learn users' passwords.

Proof: Same as the proof to Lemma 6, we only need to prove that attackers cannot receive the message from honest processes that carries the password. The honest IdP server is defined as P^i and the IdP script is defined as $script\_idp$. Here we give the proof about each processes.

- $script\_rp$. According to Algorithm 4, we can prove that RP script does not send any stored passwords.
- P^r. According to Algorithm 2, it is easy to find out that RP server does not receive or send any stored passwords.
- $script\_idp$. Based on Algorithm 3, we can find that IdP script sends the user's password at Line 68. The target of this message is Url whose host is $IdPDoaim$ set at Line 67. The $IdPDomian$ is set at Line 4 and the value is defined by the script and never modified. Therefore, the password can only be sent to the IdP server. The IdP server obtains the password at Line 10 in Algorithm 1 and does not send this parameter to any other processes.
- P^i. Based on Algorithm 1, we can find that IdP server does not send any stored passwords.

Therefore, no attackers can obtain the password from honest processes, so that this lemma is proved. ■

Lemma 2. Attacker cannot forge or modify the IdP-issued proofs.

Proof: The IdP-issued proofs include the $Cert$ used in $script\_idp$, the $RegistrationResult$ and $Token$ used in P^r . We can easily find that the IdP does not send the private key to any processes so that the attackers cannot obtain the private key. Then we only need to prove that all the proofs are well verified.

- $Cert$ is used at Line 21, 52 in Algorithm 3. At Line 21, the $Cert$ has already been verified at Line 16. At Line 52, the $Cert$ is picked from the state parameters, and the cert parameter is set at Line 19. At Line 19, the $Cert$ has already been verified at Line 16. At Line 16 the $Cert$ is verified with the public key in the scriptstate, where the key is considered initially honest and the key is not modified at Algorithm 3. Therefore, $Cert$ cannot be forged or modified.
- $RegistrationResult$ is used in Algorithm 2 from Line 35 to 55, which is verified at Line 30. The public key is initially set in the RP and never modified. Therefore, $RegistrationResult$ cannot be forged or modified.
- $Token$ is used in Algorithm 2 from Line 69 to 84 after Line 65 where it is verified. As proved before, the public key is honestly set and never modified. Therefore, $Token$ cannot be forged or modified.

Therefore, this lemma is proved. ■

Here we now show the lemma to prove that UPPRESSO meets the requirements in Definition 2 .

Lemma 3. Attacker cannot learn users' valid identity proofs.

Proof: As the $Token$ has been proved that it can not be forged by the attackers, here we only need to prove that attackers cannot receive $Token$ from other honest processes.

- Attacker cannot obtain the $Token$ from RP server. We check all the messages sent by the RP server at Line 4, 7, 19, 25, 31, 36, 45, 55, 61, 66, 74, 84 in Algorithm 2. It is easy to prove that the RP server does not send any $Token$ to other processes.
- Attacker cannot obtain the $Token$ from RP script. The messages sent by RP script can be classified into two classes. 1) The messages at Line 18, 36, 56 in Algorithm 4 are sent to the RPDomain which is set at Line 4, so that attackers cannot receive these messages. 2) The messages at Line 26, 46 only carry the contents received from RP server, and we have proved that RP server does not send any $Token$. Therefore, attackers cannot receive the $Token$ from RP script.
- Attacker cannot obtain the $Token$ from IdP server. Considering the messages at Line 4, 12, 16, 23, 26, 36, 44, 51, 67 in Algorithm 1, we find that only the message at Line 67 carries the $Token$. This $Token$ is generated at Line 65, following the trace where the $Content$ at Line 63, the PID_U at Line 61, the ID_U at Line 60, the $session$ at Line 48, and finally the $cookie$ at Line 47. That is, the receiver of $Token$ must be the owner of the $cookie$ in which session that saves the parameter ID_U . The ID_U is set at Line 15 after verifying the password and never modified. As we have already proved that the cookies and passwords cannot be known to attackers, attackers cannot obtain the $Token$ from IdP server.
- Attacker cannot obtain the $Token$ from IdP script. As the proof provided above, only IdP sends the $Token$ with the message at Line 67 in Algorithm 1, the IdP script can only receive the $Token$ at Line 99 in Algorithm 3. Here we are going to prove that the token $t(u, r)$ can only be sent to the corresponding RP server through IdP script. The receiver of $t(u, r)$ is restricted by the $RPOrigin$ at Line 100, which is set at Line 55. The host in the $RPOrigin$ is verified using the one included in $Cert$ at Line 51. If the $Cert$ belong to r, the attacker cannot obtain the $t(u, r)$. Now we give the proof that the $Cert$ belongs to r. Firstly we define the negotiated PID_{RP} in $t(u, r)$ as p. That is the PID_{RP} at Line 69 in Algorithm 2 must equal to p and the PID_{RP} is verified at Line 44 with the $RegistrationToken$. This verification cannot be bypassed due to the state check at Line 60. At the same validity period, the IdP script needs to send the registration request with same p and receive the successful registration result. As the IdP checks the uniqueness of PID_{RP} at Line 32 in Algorithm 1. The r and IdP script must share the same $RegistrationToken$. As the $RegistrationToken$ contains the $\mathtt{Hash}(N_U)$, the IdP script and r must share the same ID_{RP}. Therefore, the $Cert$ saved as the IdP scriptstate parameter must belong to r.

Therefore, attackers cannot learn users' valid identity proofs. ■

So far we have proved that UPPRESSO meets the requirements in Definition 2. Requirement 2 is satisfied.

Then, we prove that UPPRESSO meets Requirement 3. As the browser follows the same-origin policy, the attackers cannot set its cookie to user's browser at RP's origin. Therefore, due to Definition 2, the user only sets her cookie as $c(a, r)$ when RP receives the $Token$ containing the attacker's PID_U and a valid PID_{RP} negotiated by u and r. It requires that the attackers must know a valid PID_{RP}. Here we give a lemma.

Lemma 4. Attacker does not know a valid PID_{RP} negotiated by user u and RP r.

Proof: Here we give the proof that attacker cannot obtain the PID_{RP} and N_U from each processes.

- P^i. We can find in Algorithm 1, IdP only returns the message containing PID_{RP} to other processes when the PID_{RP} is included in the request message.
- P^r. Same as IdP server, RP server only sends the message containing PID_{RP} at Line 55 in Algorithm 2, and the PID_{RP} is contained in the $RegistrationResult$ received at Line 28 and verified at Line 44.
- *script_rp*. We can find in Algorithm 4, RP script only sends the messages to RP server and IdP script. The receivers' identities are ensured at Line 3, 4. Therefore, attackers cannot obtains the PID_{RP} and N_U.
- *script_idp*. The HTTP requests sent by IdP script are forwarded to the domain set at Line 4 in Algorithm 3. The HTTP requests are sent to IdP server. The postMessages are sent to the one set at Line 3, and we will prove the target cannot be attacker. According to Line 44 in Algorithm 2, PID_{RP} is valid at RP server only when RP server receives the registration result. The $\mathtt{Hash}(N_U)$ in the result ensures the result must be issued for the correct ID_{RP}. As the registration result is PID_{RP}-unique due to Line 32 in Algorithm 1, the registration result received by IdP script at Line 35 in Algorithm 3 must be same as the one in RP server. This HTTP response is related with the HTTP request at Line 28, carrying PID_{RP} and $\mathtt{Hash}(N_U)$ at Line 21, 25. It ensures the $Cert$ obtained at Line 15 must belong to RP. As the target at Line 3 is the window which opens the IdP script window and asks for user's login consent, user can easily find out the target site is not coincident with the consent requirement. Therefore, the target cannot be the attacker.

■

Therefore, Requirement 3 is satisfied and Theorem 1 is proved.

I. *Proof of Theorem 2*

In this section, we give a detailed proof about the privacy of UPPRESSO. That is, UPPRESSO is a privacy-preserving system, satisfying IdP-Privacy and RP-Privacy. As analyzed above, the requirements of IdP-Privacy and RP-Privacy are as follows.

Requirement 4. IdP-Privacy. There are honest RPs r_1, r_2, IdP i and the honest user u. We define the event sets containing each users' login procedure, for instance, the $events_{(u,r_1)}$ consists of all the events generated during the u logging in to r_1 in correct procedure. IdP-Privacy requires that for every event $e_1 \in events_{(u,r_1)}$ received by IdP, there is always an event $e_2 \in events_{(u,r_2)}$, satisfying that e_1 and e_2 are equivalent.

Here we give the proof that UPPRESSO system meets Requirement 4.

Proof: As IdP is honest, we only need to analyze the events sent to the IdP, to prove the equivalence of these events. The IdP only accepts the HTTPS requests to the path $/script, /synamicRegistration, /login, loginInfo$ and $/authorize$, which are examined as follows.

- $/script$. Based on Algorithm 1, we can find that every request to this path does not carry any parameter and body. Therefore, for event $e_1 \in events_{(u,r_1)}$ and $e_2 \in events_{(u,r_2)}$, the HTTPS messages in e_1 and e_2 meet the requirements in Definition **??**, so e_1 and e_2 are equivalent.
- $/loginInfo$. As defined in Algorithm 1, no parameters and bodies are sent to this path. The proof is same as the path $/script$.
- $/login$. According to Algorithm 1, the requests carry the body including u's username and password. For event $e_1 \in events_{(u,r_1)}$ and $e_2 \in events_{(u,r_2)}$, the usernames and passwords must be the same. Therefore, e_1 and e_2 are equivalent.
- $/dynamicRegistration$. As defined in Algorithm 1, the requests should carry the body PID_{RP}, $Endpoint$ and $Nonce$. That is, the PID_{RP} is the result of $ID_{RP}^{N_U} \mod p$, where N_U is unknown to IdP. Therefore, based on Definition 1, the PID_{RP} in e_1 and e_2 are equivalent. The $Endpoint$s and $Nonce$s are all randomly generated, therefore, they are equivalent in each events. It is proved that e_1 and e_2 are equivalent.
- $/authorize$. Based on Algorithm 1, it is easy to find that the requests to this path carry the body PID_{RP} and $Endpoint$ same as in path $/dynamicRegistration$. The proof of equivalence is also same.

Therefore, we prove that UPPRESSO meets Requirement 4. ∎

Requirement 5. RP-Privacy. There are RPs r_1, r_2, honest IdP i and the honest users u_1, u_2. The following requirements are satisfied even if r_1 and r_2 share their states.

- For every event $e_1 \in events_{(u_1,r_2)}$ received by RP, there is always an event $e_2 \in events_{(u_2,r_2)}$, and e_1 and e_2 are equivalent.
- For any user u, the event $e \in events_{(u,r_2)}$ cannot be linked with a u's $Account$ at r_1.

Here we give the proof that UPPRESSO system meets Requirement 5.

Proof: We first prove that RP-Privacy is satisfied when RPs are honest, and then extend the proof when the RPs behave malicious to use illegal parameters and conduct illegal processes.

The events known to the RPs include the postMessages sent by IdP script to RP script, and the HTTPS messages sent by RP script to RP server. However, we can find in Algorithm 2 that all the postMessages received by RP script are transmitted to RP server, so that we only need to analyze the RP's paths for HTTPS requests. For honest RP, the proof is as follows.

- $/script$. As defined in Algorithm 2, every request to this path does not carry any parameters and bodies. The HTTPS messages in event $e_1 \in events_{(u_1,r_2)}$ and $e_2 \in events_{(u_2,r_2)}$ are equivalent based on Definition **??**, so e_1 and e_2 are equivalent.
- $/login$. According to Algorithm 2, the requests to this path do not carry any parameters and bodies. Therefore, same as in path $/script$, e_1 and e_2 are equivalent.
- $/startNegotiation$. Based on Algorithm 2, we can find that the requests to this path only carry the body N_U. As N_U is a random number, the messages in e_1 and e_2 are equivalent, so e_1 and e_2 are equivalent.
- $/registrationResult$. According to Algorithm 2, the requests to this path contain the $RegistrationResult$ in the body. The $RegistrationResult$ includes PID_{RP}, $Endpoint$, $Nonce$ and the $Validity$. However, the PID_{RP} is also a random number (due to the randomness of N_U), and $Nonce$ is the hash of N_U, so PID_{RP}s and $Nonce$s are equivalent. $Endpoint$ is a random string. $Validity$ is generated based on the current time. Therefore, all the parameters in e_1 and e_2 are considered equivalent, so e_1 and e_2 are equivalent.
- $/uploadToken$. As defined in Algorithm 2, the requests to this path carry the $Token$. $Token$ consists of PID_{RP}, PID_U and $Validity$. As analyzed above, PID_{RP}s and $Validity$s in each events are equivalent. PID_U is the result of $ID_{RP}^{ID_U} \mod p$, so PID_Us are equivalent to RP who doesn't know ID_U due to Definition 1. Therefore, e_1 and e_2 are equivalent.

Moreover, with the states shared by r_2, r_1 knows all the $Account$s at r_2. However, as ID_{RP}s of r_1 and r_2 satisfy the equation $ID_{RP_{r_1}} \equiv ID_{RP_{r_2}}^x \mod p$ where x is unknown to RPs due to the discrete logarithm problem, the $Account$ at r_2 cannot be linked with the $Account$ at r_1.

Therefore, the UPPRESSO meets Requirement 5 when RPs behave honestly.

Here, we prove that UPPRESSO meets the RP-Privacy requirements even when RPs behave maliciously. The malicious RPs may attempt to steal the data from other process, or set the illegal parameters during the login procedures. That is, according to Definition 1, the PID_Us and the $Account$s must be equivalent to the attacker as long as the attacker does not know the ID_U. Based on Algorithm 1, we find that IdP does not send ID_U to any process, so PID_Us the $Account$s must be equivalent in each event.

Malicious RPs may attempt to generate $Account$ and PID_U incorrectly, however, the correct user could find this illegal behaviours as follows.

- RP may attempt to use a forged ID_{RP} or PID_{RP} to make PID_Us or $Accounts$ inequivalent. However, ID_{RP} are provided by the $Cert$, which is verified at Line 17 in Algorithm 3 using the IdP's public key that is initialized correctly and never modified. PID_{RP} is generated by the ID_{RP} at Line 21 using a nonce generated by the honest user at Line 20. Therefore, the honest user will find the illegal ID_{RP} and PID_{RP}.
- RP may attempt to make the same user upload the identity proof with same PID_U or $Account$ to break RP-Privacy. However, PID_U is generated with the nonce N_U provided by the user, and will never be controlled by the (malicious) RP. $Account$ is generated using the equation $ID_{RP}^{ID_U} \mod p$, while RPs may lead the user to use the same ID_{RP} to generate identity proof. However, the ID_{RP} is bound with $Cert$ which is verified by the user and it is easy for user to find the incorrect ID_{RP}.

Therefore, UPPRESSO system meets Requirement 5. ■

Finally, we have proved that the UPPRESSO system meets Requirement 4 and 5, and therefore Theorem 2 is proved.

Appendix B
Algorithm

A. IdP process

Algorithm 1 R^i

Input: $\langle a, f, m \rangle, s$
let $s := s'$
let $n, method, path, parameters, headers, body$ **such that** $\langle \mathtt{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$ **if** $possible$; **otherwise** stop $\langle \rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{IdPScript} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /login$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := body[username]$
  let $password := body[password]$
  if $password \not\equiv \mathtt{SecretOfID}(username)$ **then**
    let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{LoginFailure} \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $session[uid] := \mathtt{UIDOfUser}(username)$
  let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{LoginSucess} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /loginInfo$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := session[username]$
  if $username \not\equiv \mathtt{null}$ **then**
    let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{Logged} \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{Unlogged} \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /dynamicRegistration$ **then**
  let $PID_{RP} := body[PID_{RP}]$
  let $Endpoint := body[Endpoint]$
  let $Nonce := body[Nonce]$
  if $PID_{RP} \in \mathtt{ListOfPID}()$ **then**
    let $Content := \langle Fail, PID_{RP}, Nonce \rangle$
    let $Sig := \mathtt{Sig}(Content, s'.SignKey)$
    let $RegistrationResult := \langle Content, Sig \rangle$
    let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, RegistrationResult \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if
  let $Validity := \mathtt{CurrentTime}() + s'.Validity$
  let $s'.RPs := s'.RPs +^{\langle \rangle} \langle PID_{RP}, Endpoint, Validity \rangle$
  let $Content := \langle OK, PID_{RP}, Nonce, Validity \rangle$
  let $Sig := \mathtt{Sig}(Content, s'.SignKey)$
  let $RegistrationResult := \langle Content, Sig \rangle$
  let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, RegistrationResult \rangle$
  stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /authorize$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $username := session[username]$
  if $username \equiv \mathtt{null}$ **then**
    let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle \rangle, \mathtt{Fail} \rangle$
    stop $\langle f, a, m' \rangle, s'$
  end if

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

B. RP process

Algorithm 2 R^r

Input: $\langle a, f, m\rangle, s$
let $s := s'$
let $n, method, path, parameters, headers, body$ **such that** $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$ **if** $possible$; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript}\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /login$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 302, \langle\langle Location, s'.IdP.ScriptUrl\rangle\rangle, \langle\rangle\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /startNegotiation$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  let $N_U := parameters[N_U]$
  let $PID_{RP} := \texttt{ModPow}(s'.ID_{RP}, N_U, s'.IdP.p)$
  let $T := \texttt{ExEU}(N_U, s'.IdP.q)$
  let $session[N_U] := N_U$
  let $session[PID_{RP}] := PID_{RP}$
  let $session[t] := T$
  let $session[state] := expectRegistration$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle Cert, s'.Cert\rangle\rangle$
  stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /registrationResult$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.sessions[cookie]$
  if $session[state] \not\equiv expectRegistration$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    stop $\langle f, a, m'\rangle, s'$
  end if
  let $RegistrationResult := body[RegistrationResult]$
  let $Content := RegistrationResult.Content$
  if $\texttt{checksig}(Content, RegistrationResult.Sig, s'.IdP.PubKey) \equiv \texttt{FALSE}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    let $session := \texttt{null}$
    stop $\langle f, a, m'\rangle, s'$
  end if
  if $Content.Result \not\equiv OK$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    let $session := \texttt{null}$

stop $\langle f, a, m' \rangle, s'$
end if
let $PID_{RP} := session[PID_{RP}]$
let $N_U := session[N_U]$
let $Nonce := \texttt{Hash}(N_U)$
let $Time := \texttt{CurrentTime}()$
if $PID_{RP} \not\equiv Content.PID_{RP} \vee Nonce \not\equiv Content.Nonce \vee Time > Content.Validity$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
let $session := \texttt{null}$
stop $\langle f, a, m' \rangle, s'$
end if
let $session[PIDValidity] := Content.Validity$
let $Endpoint \in s'.Endpoints$
let $session[state] := expectToken$
let $Nonce' := \texttt{Random}()$
let $session[Nonce] := Nonce'$
let $Body := \langle PID_{RP}, Endpoint, Nonce' \rangle$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, Body \rangle$
stop $\langle f, a, m' \rangle, s'$
else if $path \equiv /uploadToken$ **then**
let $cookie := headers[Cookie]$
let $session := s'.sessions[cookie]$
if $session[state] \not\equiv expectToken$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
let $Token := body[Token]$
if $\texttt{checksig}(Token.Content, Token.Sig, s'.IdP.PubKey) \equiv \texttt{FALSE}$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
let $PID_{RP} := session[PID_{RP}]$
let $Time := \texttt{CurrentTime}()$
let $PIDValidity := session[PIDValidity]$
let $Content := Token.Content$
if $PID_{RP} \not\equiv Content.PID_{RP} \vee Time > Content.Validity \vee Time > PIDValidity$ **then**
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
let $PID_U := Content.PID_U$
let $T := session[t]$
let $Account := \texttt{ModPow}(PID_U, T, s'.IdP.p)$
if $Account \in \texttt{ListOfUser}()$ **then**
let $\texttt{RegisterUser}(Account)$
end if
let $session[user] := Account$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSuccess} \rangle$
stop $\langle f, a, m' \rangle, s'$
end if
stop $\langle\rangle, s'$

C. IdP scripting process

Algorithm 3 *script_idp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret \rangle$
let $s' := scriptstate$
let $command := \langle\rangle$
let $target := \texttt{PARENTWINDOW}(tree, docnonce)$
let $IdPDomain := s'.IdPDomain$
switch $s'.q$ **do**
case $start$**:**

let $N_U := \texttt{Random}()$
let $command := \langle \texttt{POSTMESSAGE}, target, \langle\langle N_U, N_U\rangle\rangle, \texttt{null}\rangle$
let $s'.Parameters[N_U] := N_U$
let $s'.q := expectCert$
case $expectCert$**:**
let $pattern := \langle \texttt{POSTMESSAGE}, *, Content, *\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
let $Cert := input.Content[Cert]$
if $\texttt{checksig}(Cert.Content, Cert.Sig, s'.PubKey) \equiv \texttt{null}$ **then**
let stop $\langle\rangle$
end if
let $s'.Parameters[Cert] := Cert$
let $N_U := s'.Parameters[N_U]$
let $PID_{RP} := \texttt{ModPow}(Cert.Content.ID_{RP}, N_U, s'.p)$
let $s'.Parameters[PID_{RP}] := PID_{RP}$
let $Endpoint := \texttt{RandomUrl}()$
let $s'.Parameters[Endpoint] := Endpoint$
let $Nonce := \texttt{Hash}(N_U)$
let $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /dynamicRegistration, \langle\rangle\rangle$
let $s'.refXHR := \texttt{Random}()$
let $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{POST},$
$\langle\langle PID_{RP}, PID_{RP}\rangle, \langle Nonce, Nonce\rangle, \langle Endpoint, Endpoint\rangle\rangle, s'.refXHR\rangle$
let $s'.q := expectRegistrationResult$
end if
case expectRegistrationResult**:**
let $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null} \wedge input.Content[RegistrationResult].type \equiv OK$ **then**
let $RegistrationResult := input.Body[RegistrationResult]$
if $RegistrationResult.Content.Result \not\equiv OK$ **then**
let $s'.q := stop$
let stop $\langle\rangle$
end if
let $command := \langle \texttt{POSTMESSAGE}, target, \langle\langle RegistrationResult, RegistrationResult\rangle\rangle, \texttt{null}\rangle$
let $s'.q := expectProofRquest$
end if
case expectProofRquest**:**
let $pattern := \langle \texttt{POSTMESSAGE}, *, Content, *\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
let $PID_{RP} := input.Content[PID_{RP}]$
let $Endpoint_{RP} := input.Content[Endpoint]$
let $s'.Parameters[Nonce] := input.Content[Nonce]$
let $Cert := s'.Parameters[Cert]$
if $Endpoint_{RP} \notin Cert.Content.Endpoints \vee PID_{RP} \not\equiv s'.Parameters[PID_{RP}]$ **then**
let $s'.q := stop$
let stop $\langle\rangle$
end if
let $s'.Parameters[Endpoint_{RP}] := Endpoint_{RP}$
let $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /loginInfo, \langle\rangle\rangle$
let $s'.refXHR := \texttt{Random}()$
let $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{GET}, \langle\rangle, s'.refXHR\rangle$
let $s'.q := expectLoginState$
end if
case expectLoginState**:**
let $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
if $input.Body \equiv \texttt{Logged}$ **then**
let $username \in ids$
let $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /login, \langle\rangle\rangle$ mystates$s'.refXHR := \texttt{Random}()$

let $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{POST}, \langle\langle username, username\rangle, \langle password, secret\rangle\rangle, s'.refXHR\rangle$
let $s'.q := expectLoginResult$
else if $input.Body \equiv \texttt{Unlogged}$ **then**
let $PID_{RP} := s'.Parameters[PID_{RP}]$
let $Endpoint := s'.Parameters[Endpoint]$
let $Nonce := s'.Parameters[Nonce]$
let $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /authorize,$
$\langle\langle PID_{RP}, PID_{RP}\rangle, \langle Endpoint, Endpoint\rangle, \langle Nonce, Nonce\rangle\rangle\rangle$
let $s'.refXHR := \texttt{Random}()$
let $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{GET}, \langle\rangle, s'.refXHR\rangle$
let $s'.q := expectToken$
end if
end if
case expectLoginResult**:**
let $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
if $input.Body \not\equiv \texttt{LoginSuccess}$ **then**
let stop $\langle\rangle$
end if
let $PID_{RP} := s'.Parameters[PID_{RP}]$
let $Endpoint := s'.Parameters[Endpoint]$
let $Nonce := s'.Parameters[Nonce]$
let $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /authorize,$
$\langle\langle PID_{RP}, PID_{RP}\rangle, \langle Endpoint, Endpoint\rangle, \langle Nonce, Nonce\rangle\rangle\rangle$
let $s'.refXHR := \texttt{Random}()$
let $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{GET}, \langle\rangle, s'.refXHR\rangle$
let $s'.q := expectToken$
end if
case expectToken**:**
let $pattern := \langle \texttt{XMLHTTPREQUEST}, Body, s'.refXHR\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
let $Token := input.Body[Token]$
let $RPOringin := \langle s'.Parameters[Endpoint_{RP}], \texttt{S}\rangle$
let $command := \langle \texttt{POSTMESSAGE}, target, \langle Token, Token\rangle, RPOrigin\rangle$
let $s.q := stop$
end if
end switch
let stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$

D. RP scripting process

Algorithm 4 *script_rp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret\rangle$
let $s' := scriptstate$
let $command := \langle\rangle$
let $IdPWindow := \texttt{SUBWINDOW}(tree, docnonce).nonce$
let $RPDomain := s'.RPDomain$
let $IdPOringin := \langle s'.IdPDomian, \texttt{S}\rangle$
switch $s'.q$ **do**
case $start$**:**
let $Url := \langle \texttt{URL}, \texttt{S}, RPDomain, /login, \langle\rangle\rangle$
let $command := \langle \texttt{IFRAME}, Url, \_SELF\rangle$
let $s'.q := expectN_U$
case $expectN_U$**:**
let $pattern := \langle \texttt{POSTMESSAGE}, *, Content, *\rangle$
let $input := \texttt{CHOOSEINPUT}(scriptinputs, pattern)$
if $input \not\equiv \texttt{null}$ **then**
let $N_U := input.Content[N_U]$
let $Url := \langle \texttt{URL}, \texttt{S}, RPDomain, /startNegotiation, \langle\rangle\rangle$

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