Contribution Title

Abstract. Single sign-on (SSO) services are widely provided in the Internet by identity providers (IdPs) as the identity management and authentication infrastructure. After authenticated by the IdP, a user is allowed to log in to relying parties (RPs) by submitting an *identity proof* (i.e., id token of OpenID Connect or SAML assertion). However, SSO introduces the potential leakage of user privacy, as (a) a curious IdP could track a user's all visits to any RPand (b) collusive RPs could link the user's identities across different RPs, to learn the user's activity profile. Existing privacy-preserving SSO solutions protect the users' activity profiles against either the curious IdP or the collusive RPs, but never prevent both of these threats.
In this paper, we propose an SSO system, called *XXX*, to protect a user's activity profile of RP visits against both the curious IdP and the collusive RPs. It separates a IdP service into two parts, the server-part service and user-part service. The server-part service runs on the IdP server, which takes the responsibility of authenticating users, storing the private key, issuing the identity token and etc. The user-part is in the user's device to prevent RP identity from being transmitted to IdP server. It generates the PPID based on the user identifier retrieved by server-part service, and transforms PPID and RP's identifier into one-time encrypted ID to prevent both IdP-based user tracing and RP-based identity linkage. To avoid the malicious user from controlling user-part service, it is protected by the Intel SGX, the secure hardware.

Keywords: SSO · Privacy · Intel SGX.

1 Introduction

Single sign-on (SSO) systems, such as OpenID Connect [23], OAuth [17] and SAML [18], have been widely deployed as the identity management and authentication infrastructure in the Internet. SSO enables a website, called the *relying party* (RP), to delegate its user authentication to a trusted third party called the *identity provider* (IdP). Thus, a user visits multiple RPs with only a single explicit authentication attempt at the IdP. With the help of SSO, a user no longer needs to remember multiple credentials for different RPs; instead, she maintains only the credential for the IdP, which will generate *identity proofs* for her visits to these RPs. SSO has been widely integrated with many application services. For example, we find that 80% of the Alexa Top-100 websites [5] support SSO, and the analysis on the Alexa Top-1M websites [13] identifies 6.30% with the SSO support. Meanwhile, many email and social network providers (such as Google, Facebook, Twitter, etc.) are serving the IdP roles in the Internet.

However, the wide adoption of SSO also raises new privacy concerns. Currently, more and more companies are interested in the users' online trace for

business practices, such as accurate delivery advertising. For example, large internet service providers, such as Google and Facebook, are interested in collecting users' online behavioral information for various purposes (e.g., Screenwise Meter [20] and Onavo [7]). Hoever, by simply serving the IdP role, these companies can easily collect a large amount of data to reconstruct users' online traces.

User privacy leaks in all existing SSO protocols and implementations. Taking a widely used SSO protocol, OpenID Connect (OIDC), as an example, we explain its login process and the risk of privacy leakage. On receiving a user's login request, the RP constructs a request of identity proof with its identity and redirects it to the IdP. After authenticating the user, the IdP generates an identify proof containing the identities of the user and the RP, which is forwarded to the RP by the user. Finally, the RP verifies the identity proof and allows the user to log in. From such login instances, any curious IdP or multiple collusive RPs could break the users' privacy as follows.

- *IdP-based login tracing.* The IdP knows the identities of the RP and user in each single login instance, to generate the identity proof. As a result, a curious IdP could discover all the RPs that the victim user attempts to visit and profile her online activities.
- *RP-based identity linkage.* The RP learns a user's identity from the identify proof. When the IdP generates identity proofs for a user, if the same user identifier is used in identity proofs generated for different RPs, malicious RPs could collude to not only link the user's login activities at different RPs for online tracking but also associate her attributes across multiple RPs.

Imagine that, a user concerning her privacy would avoid to leave her full sensitive information at an application. The user may use multiple web applications and only leave parts of her sensitive messages at each applications, for example, using real name on social website, the address on shopping website and the phone number on Telecom website. And she would try not to leave any linkable message to avoid applications combining her informations, for example, if she leave the email on each applications, they can combine the parts of informations through the email. However, the privacy leaks in SSO systems make her effort in vain. As long as a user employs the SSO system, such as Google Account, to log in to these applications, the applications providers and Google can combine your informations based on the SSO account.

Recently more and more IdPs have been considering the user's privacy serious. For example, one of the most worldwide popular instant messaging applications, WeChat, also working as the IdP service provider, enables a user to create different plain accounts for login on multiple RPs. Moreover, Active Directory Federation Services and Oracle Access Management support the use of PPID, the privacy protection scheme suggested in OIDC protocol [2, 4]; identity service providers such as NORDIC APIS and CURITY suggest adopting PPID in SSO to protect user privacy [1, 3].

However, none of the existing techniques proposed by previous research can deal with the privacy concerns comprehensively. Here we give a brief introduction

of existing solutions for privacy-preserving SSO and explain the flaws of these schemes.

- *Using different user ID in each RP.* As recommended by NIST [14] and specified in several SSO protocols [16, 23], 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.
- *Simply hiding RP ID from IdP.* For example, SPRESSO [10] were proposed to defend against IdP-based login tracing by hiding RP ID from IdP. It uses the encrypted RP ID instead, and IdP issues the identity proof for this one-time ID. However, this type of solution can only provide the same user ID for each RP.
- *Proving user identity based on zero-knowledge proof.* In this type of solution , such as EL PASSO [30] and UnlimitID [19], the user needs to keep a secret s and requires IdP to generate an identity proof for blinded s. Then user has to prove that she is the owner of s to RP without exposing s to RP. However, it is not convenient for user login on multiple devices. The user's identity is associated with the private s, therefore, if the user wants to log in to the RP on a new device, she must import the large s into this device. For security consideration, the s must be too long for user to remember.
- *Completely anonymous SSO system.* Anonymous SSO scheme is proposed to hide the user's identity to both the IdP and RPs with many methods. For example, in the anonymous SSO [15], it allows a user to visit the RP without exposing her identity to both IdP and RP based on zero-knowledge proof. However, it can only be applied to the anonymous services that do not identify the user, but not available in current personalized internet service.

As discussed above, none of the existing SSO systems defend against both IdP-based login tracing and RP-based identity linkage, and provided the convenient SSO service on multiple devices at the same time. And, it can not be solved by simply combining existing solutions together. The challenge is that, there is not a simple way for IdP to provide the RP-specific user ID without knowing the RP's identity.

In this paper, we propose XXX, which provides the convenient SSO service with comprehensive protection against bosh IdP-based login tracing and RP-based identity linkage. The key idea of XXX is separating IdP into two parts. The first part is for user authentication and identity proof issuing, which must be completed at server for security consideration. The other part is for RP and user's ID transformation, which must be completed at the user controlled and IdP trusted part to keep RP's identity unknown to IdP. With SGX, the secure hardware supported by intel CPU, the second part can be implemented at user's PC based on the *remoteattestation*, the function provided by SGX. For IdP, it can only achieve a encrypted one-time RP ID, so that the IdP-based login tracing

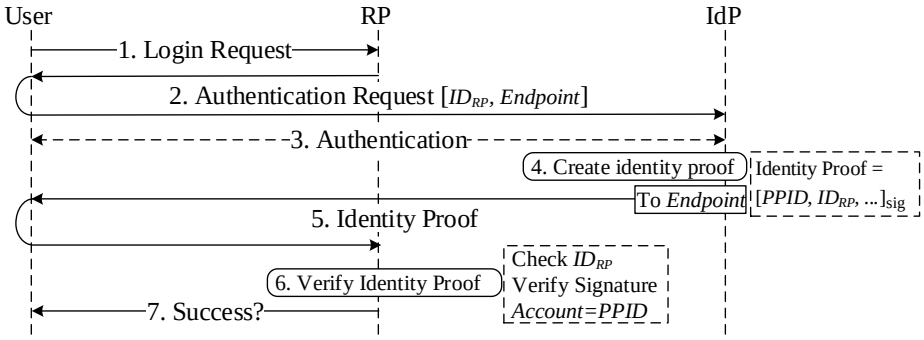

Fig. 1: The implicit protocol flow of OIDC.

is not impossible. Moreover, for RP, the user controlled IdP part generates the PPID, therefore, it protects user from RP-based identity linkage.

We summarize our contributions as follows.

- We propose the comprehensive solution to hide the users' login traces from both the curious IdP and malicious collusive RPs for convenient SSO system.
- We formally analyze the security of XXX and show that it guarantees the security, while the users' login traces are well protected.
- We have implemented a prototype of XXX, and compare the performance of the XXX prototype with the state-of-the-art SSO systems (i.e., OIDC), and demonstrate its efficiency.

The rest of the paper is organized as follows. We first introduce the background and preliminaries in Section **??**. Then, we describe the identifier-transformation based approach, the threat model, and our UPPRESSO design in Sections **??**, **??** and **??**, followed by a systematical analysis of security and privacy in Section **??**. We provide the implementation specifics and experiment evaluation in Section **??**, discuss the related works in Section **??**, and conclude our work in Section **??**.

2 Background

XXX is compatible with OIDC, and achieves the privacy protection based on the SGX. Here, we provide a brief introduction on OIDC and the SGX.

2.1 OpenID Connect

OIDC [23] is an extension of OAuth 2.0 to support user authentication, and becomes one of the most prominent SSO authentication protocols.

Implicit flow of user login. OIDC supports three processes for the SSO authentication session, known as *implicit flow*, *authorization code flow* and *hybrid flow* (i.e., a mix-up of the previous two). Here, we propose the OIDC implicit flow as the example to illustrate the protocol.

As shown in Figure 1, the implicit flow of OIDC consists of 7 steps: when a user attempts to log in to an RP (Step 1), the RP constructs a request for

identity proof, which is redirected by the user to the corresponding IdP (Step 2). The request contains ID_{RP}, RP's endpoint and a set of requested user attributes. If the user has not been authenticated yet, the IdP performs an authentication process (Step 3). If the RP's endpoint in the request matches the one registered at the IdP, it generates an identity proof (Step 4) and sends it back to the RP (Step 5). Otherwise, IdP generates a warning to notify the user about potential identity proof leakage. The RP verifies the id token (Step 6), extracts user identifier from the id token and returns the authentication result to the user (Step 7).

2.2 Intel SGX

Intel Software Guard Extensions (Intel SGX) is the hardware-based security mechanism provided by Intel since the sixth generation Intel Core microprocessors, which offers memory encryption that isolates specific application code and data in memory. It allows user-level code to allocate private regions of memory, called enclaves, which guarantees the running code are well protected from the adversary outside the enclave.

Remote Attestation. The SGX remote attestation allows a player to verify three things: the application's identity, its intactness (that it has not been tampered with), and that it is running securely within an enclave on an Intel SGX enabled platform. Moreover, with the remote attestation, the secure key exchange between the player and remote enclave application is also available even the application runs in the malicious environment.

3 Threat Model and Assumptions

XXX is compatible with OIDC, consisted of a number of RPs, user agents(i.e., the browser and enclave application) and an IdP. In this section, we describe the threat model and assumptions of these entities in XXX.

3.1 Threat Model

Adversaries' Goal: (1) The adversaries can impersonate an honest user to log in to the honest RP.

Adversaries' Capacities:

- An adversary can act as the malicious user. The adversary can control all the software running outside the enclave, for example, capturing and tempering the message transmission among enclave application, browser and IdP server, decrypting and tempering the https flow outside the enclave, tempering the script code running on the browser.
- An adversary can act as the malicious RP. The adversary can lead the user to log in to the malicious RP. In this situation, the adversary can manipulate or all the message transmitted through RP, and collect all the flows received from user to link the user identity.
- An adversary can act as the curious but honest IdP. The curious IdP can store and analyze the received messages, and perform the timing attacks, attempting to achieve the IdP-based linkage. However, the honest IdP must process the requests of RP registration and identity proof correctly, and never colludes with others (e.g., malicious RPs and users).

– Moreover, an adversary can collect all the network flow transmitted through a user. The adversary can also lead the user download the malicious script on her browser.

3.2 Assumptions

We assume the honest user's device is secure, for example, the user would not install any malicious application on her device. The application and data inside the enclave are never tempered or leaked, even in the malicious user's device.

The TLS is also adopted and correctly implemented at the system, so that the communications among entities ensure the confidentiality and integrity. The cryptographic algorithms and building blocks used in XXX are assumed to be secure and correctly implemented.

Phishing attack is not considered in this paper.

4 Design of XXX

The XXX is compatible with OIDC, besides that the IdP service is separated into user agent part and server part. The server part IdP service takes the responsibility of authenticating the user, retrieves the UID for each user, and issues the signed identity proof consisted the privacy-preserving RP and user identifier generated at user agent part IdP service. The user agent part IdP service would obtain the UID from server part service, transform UID into the PPID, and encrypt the RPID and PPID with an one-time symmetric key to avoid IdP server digging out the RP's identity. As the enclave application is protected by SGX, it must not conduct any malicious behaviour.

4.1 XXX process

In this section, we provide the detail protocol of XXX.

The process of XXX is depicted in detail in Figure 2. The SSO process is started with the user's visit to an RP at her browser, and the browser downloads the RP script (step 1), which is used to conduct the behaviour defined by RP at user side. Then the RP script opens the new window with the RP login endpoint (step 2, 3). Then the user is redirected to the IdP server (step 4). It must be noticed that, the user cannot visit IdP at step 2,3 directly because of the $Referer$ attribute in HTTP header. While the script in origin A opens a new window with origin B, the HTTP request to B will carry the key value $Referer : A$. Therefore, the RP's domain is exposed to IdP. With HTML5, a special attribute for links in HTML was introduced, that the $ref =$ "$noreferrer$" can be used to make $Referer$ header be suppressed. However, when such a link is used to open a new window, the new window does not have a handle on the opening window (opener) anymore. The handle is necessary for XXX to transmit messages between RP and IdP.

While the user is redirected to the IdP server, the IdP will retrieve the IdP script (step 5), witch is used to deal with the interaction with IdP server, RP script and enclave application. And the IdP authenticates the user (step 6). After the IdP script is downloaded, it sends the ready signal to its opener (i.e., the RP window) (step 7). Then it will receive the $RPdomain$ from RP script for

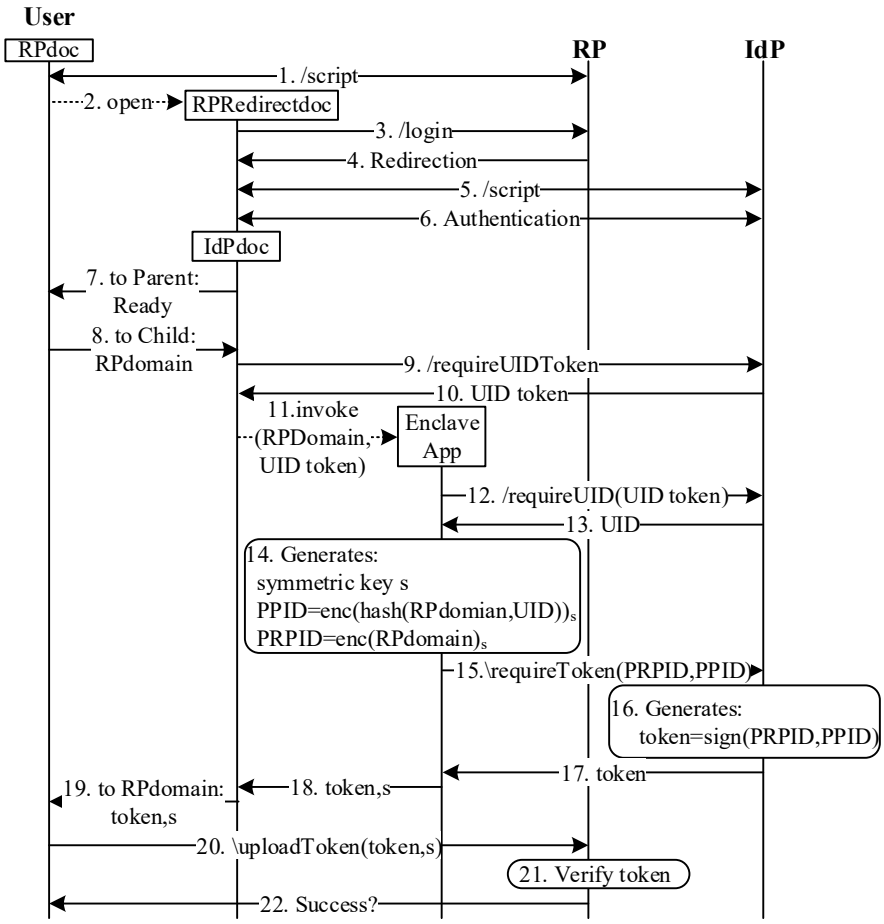

Fig. 2: The protocol flow of XXX.

further process (step 8). There are some types of parameters required in OIDC protocol to be carried in the SSO request, such as *response_type* and *scope*. In this paper, we would not focus on these attributes, and only describe the necessary parameters.

Then the user starts the SSO request to IdP (step 9). As the result, the user receives a $UIDtoken$ from IdP (step 10), with which an enclave application can retrieve the user's real identifier UID from IdP server.

Then the IdP script invokes the enclave application with RP's domain and the $UIDtoken$ (step 11). The enclave application requires the UID from IdP server with the UID token (step 12, 13). After receiving the UID, enclave application generates the symmetric key s, encrypts the RP's domain with key s as the transformed RP ID, and encrypts the hash of RP's domain and UID as the PPID (step 14).

After the ID transformation, enclave application requires the IdP server to generate identity proof with $PRPID$ and $PPID$ (step 15). The IdP server signs the *token* consisted of $PRPID$ and $PPID$ as the identity proof (step 16), and return it to enclave application (step 17). Then enclave application set the *token* and key s as the result of step 11 (step 18). The IdP script then sends the *token* and s to the origin $RPdomain$ through $postMessage$ (step 19). It guarantees that only the script running in the RP window can receive the *token*, which avoids the man-in-the-middle attack.

Finally, the RP script uploads the *token* and key s to RP server (step 20). The RP server firstly verifies the signature with IdP's public key, then generates the $PRPID$ with its domain and key s, and compared it with the one carried by *token*. If the two $PRPID$s are equal, RP decrypts the user's ID from $PPID$, and find out the related user information in its database (step 21). Then it returns the login result to user (step 22).

5 Security Analysis

Our formal analysis of XXX is based on the Dolev-Yao style web model [10], which has been widely used in formal analysis of SSO protocol, e.g., OAuth 2.0 [11] and OIDC [12]. To make the description cleaner, we foucus on our communications among XXX entities, and assume DNS and HTTPS are secure, which has already been anaylzed in [10].

5.1 The Web Model

The main entities in the model are *atomic processes*, which represent the essential nodes in the web systems, such as browsers, web servers and attackers. The atomic processes communicate with each other through the *events* containing the receiver atomic process's address (IP), the sender atomic process process's address (IP) and the transmitted *messages*. Moreover, there are also dependent *scripting processes* which runs on the client-side environment relying on the browsers such as JavaScript. The scripting provides the server defined function to the browser. The web system mainly consists of the set of atomic processes and scripting processes. The operation of a system is described as that the system converts its states via step of runs. The state of web system is called $configuraton$ which consists of all the states of the atomic processes in the system and all the event can be accepted by the processes.

Here, we list the definitions of these notations as follows.

Message is defined as formal terms without variables (called ground terms). The messages in the model is considered containing constants (such as ASCII strings and nonce), sequence symbols (such as n-ary sequences $\langle\rangle$, $\langle . \rangle$, $\langle ., . \rangle$ etc.) and further function symbols (such as encryption/decryption and digital signatures). For example, an HTTP request is a common message in the web model, containing a type `HTTPReq`, a nonce n, a method `GET` or `POST`, a domain , a path, URL parameters, request headers, and the body in the sequence symbol formate. Here is an example for an HTTP GET request for the domain

`exa.com/path?para = 1` with the headers and body empty.

$$m := \langle \texttt{HTTPReq}, n, \texttt{GET}, exa.com, /path, \langle\langle para, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

Event is the basic communication elements in the model. An event is the term in the formate $\langle a, f, m\rangle$. In an *event*, the f is the sender's address, the a represents the address receiver, and m is the message transmitted.

Atomic Process. An *atomic* $Dolev-Yao(DY)$ *process* is can be displayed as the tuple $p = (I^p, Z^p, R^p, s_0^p)$, which stands for the single node in the web model, such as the server and browser. I^p includes the addresses owned by this process. Z^p is the set of states that this process is probably in. R^p is the set of relations between the pairs $\langle s, e\rangle$ and $\langle s', e'\rangle$ where $s, s' \in Z^p$. That is, once a process is in the sate s and receives the event e, it would jump into the state s' and wait for the event e'.

Browser Process. In XXX, we assume that the browsers are honest, therefore, we only need to analyze how the browsers interactive with the scripts. The browser model mainly includes the windows and documents.

- **Window**. The window w represents the the concrete browser window in the system, which is identified by a *nonce*. A window contains a set of `documents` including the current document and cached documents.
- **Document**. The document is the HTML content in a window, which is identified by the document *nonce*. It includes the scripting process downloaded from server.

Scripting Process. The web model also contains the scripting process representing the client-side script loaded by browser such as JavaScript code. However, the *scripting process* must rely on an *atom process* such as browser and provide the relation R witch is called by this *atomic process*.

Equational Theory is defined as usual in Dolev-Yao models, but introduces the symbol $\equiv$ representing the congruence relation on terms. For instance, $dec(enc(m,\ k),\ k) \equiv m$, where k is the symmetric key.

Web System. The web system is consisted of a set of processes (including atomic processes and scripting processes). The web system can be described as the tuple ($\mathcal{W}$, $\mathcal{S}$, `script`, E^0). $\mathcal{W}$ is consisted of the atomic processes, including honest processes and malicious processes. $\mathcal{S}$ is the set of scripting processes including honest scripts and malicious scripts. `script` is the set of concrete script code. And E^0 includes all the events that could be accepted by the processes in $\mathcal{W}$.

Configuration. In the web system, there is the set of states of all processes in $\mathcal{W}$ at one point in time, denoted as S. And all the *event*s can be accepted by the processes at this point consist the set E. A $configuration$ of the system is defined as the tuple (S, E, N) where N is the mentioned sequence of unused nonces.

Run Step. A run step is the process, that a web system changes its configuration (S, E, N) into (S', E', N') after accepting an event $e \in E$.

5.2 Model Of XXX

The XXX model is a web system which is defined as

$$\mathcal{XWS} = (\mathcal{W}, \mathcal{S}, \texttt{script}, E^0),$$

$\mathcal{W}$ is the finite set of atom processes in XXX system including a single IdP server process, multiple honest RP server processes, the browser processes, the enclave application processes and the attacker processes. We assume that all the honest RPs are implemented following the same rule so that the process are considered consistent besides of the addresses they listen to. The browsers controlled by user are considered honest. That is, the browser controlled by attackers can behave as an independent atomic process. The enclave applications are always honest.

$\mathcal{S}$ is the finite set of scripting processes consists of *script_rp*, *script_idp* and *script_attacker*. The *script_rp* and *script_idp* are downloaded from honest RP and IdP processes and the *script_attacker* is downloaded from attacker process considered existing in all browser processes.

5.3 Security of XXX

In this section, we prove the security of XXX. Here, we give the theorem to be proved.

Theorem 1. *Let $\mathcal{XWS}$ be the XXX web system, then $\mathcal{EWS}$ is secure.*

The XXX is considered secure iff the adversary cannot log in to an honest RP under another honest user's account. Based on the model of XXX, described in Appendix A, only when the RP accepts an valid identity token and retrieves the PPID same as the honest user's from adversary, the attack is successful. Therefore, we can get the definition.

Definition 1. *Let $\mathcal{XWS}$ be an XXX web system, $\mathcal{XWS}$ is secure* iff *the adversary cannot obtain a valid identity token, whose encrypted PPID could be decrypted into the honest user's PPID.*

To prove XXX system is secure, we only need to guarantee that it satisfies the requirements described in definition 1. The adversary may try to retrieve a valid identity token in following ways: (1) forging the valid identity token itself; (2) leading the honest RP treat adversary's identity token as the honest user's one; (3) stealing a valid token from an honest entity in the system. Therefore, definition 1 can be further classified into the following lemmas.

Lemma 1. *The adversary cannot forge the valid identity token.*

Proof. It can be easily proved that, while an RP receives the identity token, it verifies the signature with the IdP issued public key. Only the IdP knows the corresponding private key, so that the adversary cannot forge the valid identity token itself.

Lemma 2. *The RP would not retrieve an honest user's PPID from the adversary's identity token.*

Proof. The adversary cannot obtain the identity proof issued for other honest user from XXX system, which is to be proved in lemma 3. We consider in the SSO process, only the IdP server and enclave application are honest. An adversary can get a identity token including $\texttt{PRPID} = encrypt(\texttt{Domain})$ and $\texttt{PRPID} = encrypt(hash(\texttt{Domain}, \texttt{uid}))$. The `Domain` is controlled by adversary and can be assigned as any value. The `uid` must belong to the adversary. There is no chance that an adversary can get a key, making the attack available. Because the key must satisfies that, $decrypt(\texttt{PRPID}, key) \equiv honest\ RPDomain$ and $decrypt(\texttt{PPID}, key) \equiv hash(honest\ RPDomain, honest\ uid)$, which is not possible.

Lemma 3. *The adversary cannot steal an identity token from any entities in the system.*

Proof. Firstly, we give the brief description of the proof. (1) For IdP, it only sends the identity token to its enclave application. (2) For enclave application, it only returns the token back to whom invoked it with the uid token. And it can be proved that only the honest user can own the uid token beside of enclave application and IdP server. (3) The IdP script only transmits the token to the script in the origin `RPDomain`. As the identity token includes the `PRPID`, the identity proof would be only sent to the origin that the token is issued for. (4) The RP script only sends the identity token to its server. (5) RP sever would not send identity token to any parties. The detailed proof is described as follows.

The identity token sent by IdP is shown in line 44, Algorithm 1, as the response of path described in line 38, Algorithm 1. We consider that IdP only accepts the enclave application's request to path `requireUID` and `requireToken`. Therefore, the identity token would only be sent to the enclave application.

We can see that the identity token is sent by enclave application shown in line 25, Algorithm 5 to the entity defined in line 2, 5. The receiver of identity token is the one invoking it with a uid token (line 4, Algorithm 5). And the uid is exchanged with this uid token (line 9, 14, Algorithm 5). Then we prove that the only the honest user can own the uid token. Based on line 11-16 and line 22-28, Algorithm, we can see that IdP only sends the uid token to the authenticated user. It can be observed that the IdP script receives the uid token in line 34, Algorithm 3 and only sends it to enclave application in line 35, Algorithm 3. Therefore, an adversary cannot obtain a user's uid token, so that it cannot retrieve the identity token from the enclave application.

The IdP script only sends the identity token by postMessage shown at line 43, Algorithm 3. The target is the opener of this window and restricted by the origin $mathttRPDomain$. `RPDomain` is defined at line 25, Algorithm 3 and never rewrote. Due to the scriptstate, in Algorithm 3 the `RPDomain`s used at line 35 (used for identity token generation) and line 43 (restricting the receiver) must be consistent with the one at line 25. Therefore, IdP would not send the identity token to any adversaries.

The model Algorithm 4 shows RP script receives the identity token at line 21 and send it out at line 25, while the receiver is defined at line 23. The receiver address is assigned during initiation at line 4 and never modified. It is downloaded with the script and considered honest. Therefore, the RP script would not send the identity token to adversary.

Based on the model shown as Algorithm 2, we can find that RP server would not send identity token to other parties. Therefore, the adversary cannot steal the identity token from RP server.

It is proved that XXX system satisfies the requirements raised in definition 1. Therefore, Theorem 1 is proved.

5.4 Privacy of XXX

Based on the process shown in Section 4 and models shown in Appendix A, we can find that a curious IdP can only obtain the `PRPID` related with the RP's identity. The `PRPID` is the transformed RP domain, which is encrypted with an one-time key. Therefore, the IdP cannot know the real RP's identity or link multiple logins on the same RP. So the IdP-based identity tracing is not possible in XXX system.

Similarly, the RP can only obtains the `PPID` from IdP. An malicious RP cannot derive the real user's uid from the $hash(\texttt{RPDomain}, \texttt{uid})$. The collusive RPs are also unable to link the same user because of the same reason. Although the malicious RP can control the `RPDomain` to lead the IdP generate incorrect `PPID`, it still fails to accomplish the attack. Because due to the proof of lemma 3, once the `RPDomain` is not consist with the RP's origin, the RP script would be unable to receive the identity token. Therefore, the attack is not available.

However, the attack based on user's cookie is not considered in this paper. That is, the cookie of user may be exploited by malicious RPs to link a user at different RPs. For example, a user may visit multiple RPs at the same time. The malicious RPs may redirect the user to each other through the hidden iframe, carrying the `PPID`. This attack is also available in other user authentication systems beside of SSO system. Moreover, it can be easily detected through multiple methods, such as checking the iframe in the script, observing redirection flow through browser network tool, and detecting the redirection based on the browser extension.

6 Implementation and Evaluation

6.1 Implementation

We have implemented a prototype of XXX, containing an IdP server, an RP server and the enclave application.

IdP server.

RP server.

Enclave application.

6.2 Evaluation

7 Related Works

Recently, SSO systems have been the essential infrastructure of internet service. Various SSO protocols, such as OAuth, OIDC and SAML are widely adopted by Google, Facebook and other companies. There are also troubles about the security and privacy are introduced with the SSO systems.

Various attackers were found able to access the honest user's account at RP by multiple methods. Some attackers use the stolen user's identity proof (or cookie) to impersonate a user, while the attacks are based on the vulnerabilities of user's platforms [6,27]. Some other attackers try to temper the identity token to impersonate an honest user, such as XML Signature wrapping (XSW) [24], RP's incomplete verification [22, 25, 27], IdP spoofing [21, 22] and etc. In some IdP services did not bind the issued identity token with a corresponding RP (or the honest RP did not verify the binding), so that a malicious RP can leverage the received identity token to log in to another RP [21, 22, 25, 28, 28]. Moreover, automatical tools, such as SSOScan [31], OAuthTester [29] and S3KVetter [28], are also designed to detect vulnerabilities in SSO systems.

Beside of the analysis on the implemented SSO systems, the formal analysis is also used to guarantee the security of SSO protocols. Fett et al. [11, 12] analyse the OAuth 2.0 and OIDC protocols based on the expressive Dolev-Yao style model [8]. The vulnerabilities found in these analysis enable the adversary steal the identity token form SSO systems. The analysis also shows that OAuth 2.0 and OIDC are secure once these two vulnerabilities prevented.

NIST [14] suggests that the SSO should protect user from both RP-based identity linkage and IdP-based login tracing. The pairwise user identifier (e.g., PPID) is used in some SSO protocols, such as SAML [18] and OIDC [23]. Some protocols simply hide the RP's identity from IdP, such as SPRESSO [10] (using encrypted RP identifier) and BrowserID [9] (RP identifier is added by user). However, the pairwise user identifier cannot prevent IdP-based login tracing, and simply hiding RP identifier is not defensive to RP-based identity linkage. Moreover, an analysis on Persona found IdP-based login tracing could still succeed [8, 9]. There is also another method that prevents both IdP and RP from knowing user's trace based on zero-knowledge algorithm, such as EL PASSO [30] and UnlimitID [19]. However, this type of solution requires that the user must remember a secret representing her identity, which is not convenient for login on multiple devices.

Anonymous SSO schemes enable the users to access a service (i.e. RP) with the permission from a verifier (i.e., IdP) without revealing their identities. The anonymous SSO systems are designed based on multiple methods. For example, various cryptographic primitives, such as group signature, zero-knowledge proof, etc., were used to design anonymous SSO schemes [15, 26]. The anonymous SSO schemes allow a user to obtain a unidentified token anonymously, and access RP's service with this token. However, the RP cannot classify a user's multiple logins, as the user is completely anonymous in this system. Therefore, they are not available in current personalized internet service.

8 Conclusion

In this paper, we propose XXX, the XXX. To prevent the IdP from knowing a user's login trace in an SSO system, we split a IdP service into two parts. One part runs on the IdP server, which takes the responsibility of authenticating users, storing the private key, issuing the identity token and etc. The other part generates the PPID based on the user identifier retrieved by server-part service, and transforms PPID and RP's identifier into one-time encrypted ID to prevent both IdP-based user tracing and RP-based identity linkage. This part of service runs on the user's device, but protected by the Intel SGX. Even the malicious cannot control the user-part service. We formally analyse the XXX protocol based on Dolev-Yao style model, and make sure its security. The performance evaluation on the prototype of XXX demonstrates that the introduced is modest.

References

1. Build gdpr compliant apis with openid connect. `https://nordicapis.com/build-gdpr-compliant-apis-with-openid-connect/`, accessed August 20, 2020
2. Introducing identity federation in oracle access management. `https://docs.oracle.com/cd/E40329_01/admin.1112/e27239/oif_1.htm`, accessed August 20, 2020
3. Pairwise pseudonymous identifiers. `https://curity.io/resources/architect/openid-connect/ppid-intro/`, accessed August 20, 2020
4. The role of claims. `https://docs.microsoft.com/en-us/windows-server/identity/ad-fs/technical-reference/the-role-of-claims`, accessed August 20, 2020
5. The top 500 sites on the web. `https://www.alexa.com/topsites`, accessed July 30, 2019
6. Armando, A., Carbone, R., Compagna, L., Cuéllar, J., Pellegrino, G., Sorniotti, A.: An authentication flaw in browser-based single sign-on protocols: Impact and remediations. Computers & Security **33**, 41–58 (2013)
7. Cyphers, B., Kelley, J.: What we should learn from "facebook research". `https://www.eff.org/deeplinks/2019/01/what-we-should-learn-facebook-research`, accessed July 20, 2019
8. Fett, D., Küsters, R., Schmitz, G.: 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. pp. 673–688 (2014)
9. Fett, D., Küsters, R., Schmitz, G.: 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). pp. 43–65 (2015)
10. Fett, D., Küsters, R., Schmitz, G.: 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. pp. 1358–1369 (2015)
11. Fett, D., Küsters, R., Schmitz, G.: 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. pp. 1204–1215 (2016)
12. Fett, D., Küsters, R., Schmitz, G.: 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. pp. 189–202 (2017)
13. Ghasemisharif, M., Ramesh, A., Checkoway, S., Kanich, C., Polakis, J.: 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. pp. 1475–1492 (2018)
14. Grassi, P.A., Garcia, M., Fenton, J.: NIST special publication 800-63c digital identity guidelines: Federation and assertions. National Institute of Standards and Technology, Los Altos, CA (2017)
15. Han, J., Chen, L., Schneider, S., Treharne, H., Wesemeyer, S.: 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. pp. 470–490 (2018)
16. Hardjono, T., Cantor, S.: SAML v2.0 subject identifier attributes profile version 1.0. OASIS standard (2019)
17. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC **6749**, 1–76 (2012)
18. Hughes, J., Cantor, S., Hodges, J., Hirsch, F., Mishra, P., Philpott, R., Maler, E.: Profiles for the OASIS Security Assertion Markup Language (SAML) v2.0. OASIS standard (2005), accessed August 20, 2019
19. Isaakidis, M., Halpin, H., Danezis, G.: Unlimitid: Privacy-preserving federated identity management using algebraic macs. In: Weippl, E.R., Katzenbeisser, S., di Vimercati, S.D.C. (eds.) Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2016, Vienna, Austria, October 24 - 28, 2016. pp. 139–142. ACM (2016)
20. Li, S., Kelley, J.: 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
21. Mainka, C., Mladenov, V., Schwenk, J.: 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. pp. 321–336 (2016)
22. Mainka, C., Mladenov, V., Schwenk, J., Wich, T.: 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. pp. 251–266 (2017)
23. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C.: OpenID Connect core 1.0 incorporating errata set 1. The OpenID Foundation, specification (2014)
24. Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., Jensen, M.: On breaking SAML: Be whoever you want to be. In: Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA. pp. 397–412 (2012)
25. Wang, H., Zhang, Y., Li, J., Gu, D.: 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. pp. 167–176 (2016)
26. Wang, J., Wang, G., Susilo, W.: 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. pp. 560–567 (2013)
27. Wang, R., Chen, S., Wang, X.: 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. pp. 365–379 (2012)

28. Yang, R., Lau, W.C., Chen, J., Zhang, K.: Vetting single sign-on SDK implementations via symbolic reasoning. In: Proceedings of the 27th USENIX Security Symposium, USENIX Security, Baltimore, MD, USA. pp. 1459–1474 (2018)
29. Yang, R., Li, G., Lau, W.C., Zhang, K., Hu, P.: 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. pp. 651–662 (2016)
30. Zhang, Z., Król, M., Sonnino, A., Zhang, L., Rivière, E.: EL PASSO: efficient and lightweight privacy-preserving single sign on. Proc. Priv. Enhancing Technol. **2021**(2), 70–87 (2021)
31. Zhou, Y., Evans, D.: SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA. pp. 495–510 (2014)

A Appendix: Web Model

A.1 Elements of Model

Here we provide the details of the format of the messages we use to construct the XXX 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 mapping response to corresponding request.
- *method* is 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 $\langle \langle name, value \rangle \rangle$, such as $\langle \langle Referer, http://example.com \rangle, \langle Cookies, c \rangle \rangle$.
- *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.

POSTMESSAGE. PostMessage is used in the browser for transmitting messages between scripts from different origins. We define the sent 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. However, the received postMessage is defined as $\langle \texttt{POSTMESSAGE}, sender, Content, Origin \rangle$. The `sender` is the origin of the postMessage of sender.

XMLHTTPREQUEST. XMLHTTPRequest is the HTTP message transmitted by scripts in 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.

ENCLAVEMESSAGE. EnclaveMessage is the message transmitted between Script Process and Enclave Application. It is defined as the term $\langle \texttt{ENCLAVEMESSAGE}, Content \rangle$.

Data Operation. The data used in ExtraF 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 $body[password] \equiv 123$.

Scripting Process. 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 XXX system. We have defined the postMessage and XMLHTTPRequest (for HTTP request) message which are the *commands*. Moreover, a term in the form $\langle$`IFRAME`$, URL, WindowNonce\rangle$ asks the browser to create this document's subwindow and it visits the server with the URL.

A.2 XXX Model

IdP server. The state of IdP server is a term in the form $\langle sessions, Users, Tokens, SignKey\rangle$. Other data stored at IdP but not used during SSO process is not mentioned here. Moreover, we consider the communications between enclave application and server, such as remote attestation, key exchange and trustful channel, are well implemented. So they are not displayed in the model.

- **sessions** is the term in the form of $\langle\langle Cookie, session\rangle\rangle$, the Cookie uniquely identifies the session and session stores the messages uploaded by the browser.
- **Users** is the set of all users' information, containing the *username*, *password*, *uid*, and so on.

– **Tokens** contains mappings between the issued uid token and uid.
– **SignKey** is IdP's private used for generating identity token.

Algorithm 1 R^i

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 $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{IdPScript}\rangle$
    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 $password \not\equiv \texttt{SecretOfID}(username)$ **then**
        let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure}\rangle$
        stop $\langle f, a, m'\rangle, s'$
    let $session[uid] := \texttt{UIDOfUser}(username)$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess}\rangle$
    stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /requireUIDToken$ **then**
    let $cookie := headers[Cookie]$
    let $session := s'.sessions[cookie]$
    let $uid := session[uid]$
    if $uid \equiv \texttt{null}$ **then**
        let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{UnLogged}\rangle$
        stop $\langle f, a, m'\rangle, s'$
    let $token := \texttt{GenerateToken}()$
    let $s'.Tokens := s'.Tokens +^{\langle\rangle} \langle uid, token\rangle$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{Token}, token\rangle\rangle$
    stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /requireUID$ **then**
    let $UIDToken := body[UIDToken]$
    let $uid := \texttt{FindUIDByToken}(\texttt{UIDToken})$
    if $uid \equiv null$ **then**
        let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{TokenError}\rangle$
        stop $\langle f, a, m'\rangle, s'$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{UID}, uid\rangle\rangle$
    stop $\langle f, a, m'\rangle, s'$
else if $path \equiv /requireToken$ **then**
    let $PRPID := body[PRPID]$
    let $PPID := body[PPID]$

41: **let** $Content := \langle PRPID, PPID \rangle$
42: **let** $token := Content + Sign(Content, s'.SignKey)$
43: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{Token}, token \rangle \rangle$
44: **stop** $\langle f, a, m' \rangle, s'$
45: **stop** $\langle\rangle, s'$

RP server. The state of RP server is a term in the form $\langle sessions, IdP, Users, RPDomain \rangle$.

- **IdP** is the set the information of IdP server, including the public key, domain, and so on.
- **Users** is the set of all users' information, containing the $PPID$ and other necessary attributes.
- **RPDomain** is RP's address.

Algorithm 2 R^r

Input: $\langle a, f, m \rangle, s$
1: **let** $s' := s$
2: **let** $n, method, path, parameters, headers, body$ **such that**
3: $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body \rangle \equiv m$
4: **if** *possible*; **otherwise** stop $\langle\rangle, s'$
5: **if** $path \equiv /script$ **then**
6: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript} \rangle$
7: **stop** $\langle f, a, m' \rangle, s'$
8: **else if** $path \equiv /uploadToken$ **then**
9: **let** $cookie := headers[Cookie]$
10: **let** $session := s'.sessions[cookie]$
11: **let** $token := body[Token]$
12: **let** $key := body[Key]$
13: **if** $\texttt{CheckSig}(Token.Content, Token.Sig, s'.IdP.PubKey)$ **then**
14: **let** $PRPID := \texttt{Encrypt}(s'.RPDomain, key)$
15: **if** $PRPID \equiv token.Content.PRPID$ **then**
16: **let** $PPID := \texttt{Decrypt}(token.Content.PPID, key)$
17: **if** $PPID \notin \texttt{ListOfUser}()$ **then**
18: **let** $\texttt{RegisterUser}(PPID)$
19: **let** $session[user] := PPID$
20: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSuccess} \rangle$
21: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail} \rangle$
22: **stop** $\langle f, a, m' \rangle, s'$
23: **stop** $\langle\rangle, s'$

IdP script. The state of IdP script is a term in the form $\langle IdPDomain, Parameters, q, refXHR \rangle$.

- $IdPDomain$ is the IdP's host.
- $Parameters$ is used to store the parameters received from other processes.
- q is used to label the procedure point in the login.

– $refXHR$ is the nonce to map HTTP request and response.

Algorithm 3 *script_idp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret\rangle$

```
1:  let s' := scriptstate
2:  let command := ⟨⟩
3:  let target := PARENTWINDOW(tree, docnonce)
4:  let IdPDomain := s'.IdPDomain
5:  switch s'.q do
6:      case startLogin:
7:          let username ∈ ids
8:          let Url := ⟨URL, S, IdPDomain, /login, ⟨⟩⟩
9:          let s'.refXHR := Random()
10:         let command := ⟨XMLHTTPREQUEST, Url, POST,
            ⟨⟨username, username⟩, ⟨password, secret⟩⟩, s'.refXHR⟩
11:         let s'.q := expectLoginResult
12:     case expectLoginResult:
13:         let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
14:         let input := CHOOSEINPUT(scriptinputs, pattern)
15:         if input ≢ null then
16:             if input.Body ≢ LoginSuccess then
17:                 let stop ⟨⟩
18:             let command := ⟨POSTMESSAGE, target, Ready, null⟩
19:             let s'.q := expectRPDomain
20:     case expectRPDomain:
21:         let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
22:         let input := CHOOSEINPUT(scriptinputs, pattern)
23:         if input ≢ null then
24:             let RPDomain := input.Content[RPDomain]
25:             let s'.Parameters[RPDomain] := RPDomain
26:             let Url := ⟨URL, S, IdPDomain, /requireUIDToken, ⟨⟩⟩
27:             let s'.refXHR := Random()
28:             let command := ⟨XMLHTTPREQUEST, Url, GET, s'.refXHR⟩
29:             let s'.q := expectUIDToken
30:     case expectUIDToken:
31:         let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
32:         let input := CHOOSEINPUT(scriptinputs, pattern)
33:         if input ≢ null then
34:             let token := input.Body[UIDToken]
35:             let command := ⟨ENCLAVEMESSAGE, ⟨⟨UIDToken, token⟩,
                ⟨RPDomain, s'.Parameters[RPDomain]⟩⟩⟩
36:             let s'.q := expectIdentityToken
37:     case expectIdentityToken:
38:         let pattern := ⟨ENCLAVEMESSAGE, Content⟩
```

39: **let** $input$:= CHOOSEINPUT($scriptinputs, pattern$)
40: **if** $input \not\equiv$ `null` **then**
41: **let** $token := input.Content[Token]$
42: **let** $key := input.Content[Key]$
43: **let** $command := \langle$`POSTMESSAGE`$, target, \langle\langle$`IdentityToken`$, token\rangle, \langle$`Key`$, key\rangle\rangle,$ `s'.Parameters[RPDomain]`$\rangle$
44: **let** $s'.q := stop$
45: **stop** $\langle s', cookies, localStorage, sessionStorage, command\rangle$

RP script. 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.

Algorithm 4 *script_rp*

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secret\rangle$
1: **let** $s' := scriptstate$
2: **let** $command := \langle\rangle$
3: **let** $IdPWindow$:= SUBWINDOW($tree, docnonce$).$nonce$
4: **let** $RPDomain := s'.RPDomain$
5: **switch** $s'.q$ **do**
6: **case** *start*:
7: **let** $Url := \langle$`URL`$, $`S`$, RPDomain, /login, \langle\rangle\rangle$
8: **let** $command := \langle$`IFRAME`$, Url, \_SELF\rangle$
9: **let** $s'.q := expectReady$
10: **case** *expectReady*:
11: **let** $pattern := \langle$`POSTMESSAGE`$, s'.IdPDomain, Content, *\rangle$
12: **let** $input$:= CHOOSEINPUT($scriptinputs, pattern$)
13: **if** $input \not\equiv$ `null` $\wedge\ input.Content \equiv$ `Ready` **then**
14: **let** $Content := \langle$`RPDomain`$, RPDomain\rangle$
15: **let** $command := \langle$`POSTMESSAGE`$, target, Content,$ `null`$\rangle$
16: **let** $s'.q := expectToken$
17: **case** *expectToken*:
18: **let** $pattern := \langle$`POSTMESSAGE`$, s'.IdPDomain, Content, *\rangle$
19: **let** $input$:= CHOOSEINPUT($scriptinputs, pattern$)
20: **if** $input \not\equiv$ `null` **then**
21: **let** $token := input.Content[Token]$
22: **let** $key := input.Content[Token]$
23: **let** $Url := \langle$`URL`$, $`S`$, RPDomain, /uploadToken, \langle\rangle\rangle$
24: **let** $s'.refXHR$:= `Random`()
25: **let** $command := \langle$`XMLHTTPREQUEST`$, Url,$ `POST`$,$ $\langle\langle$`Token`$, token\rangle, \langle$`Key`$, key\rangle\rangle, s'.refXHR\rangle$
26: **let** $s'.q := stop$
27: **stop** $\langle s', cookies, localStorage, sessionStorage, command\rangle$

Enclave application. The state of enclave application is a term in the form $\langle IdP, Parameters \rangle$.

- IdP contains the IdP's information, including $Host$, $path$, and so on.
- $Parameters$ is used to store the parameters received from other processes.

Algorithm 5 R^e

Input: $\langle a, f, m \rangle, s$

let $s' := s$
let $\texttt{InvokeFrom} := f$
if $m.type \equiv \texttt{ENCLAVEMESSAGE}$ **then**
  let $token := m.Content[Token]$
  let $RPDomain := m.Content[RPDomain]$
  let $s'.Parameters[RPDomain] := RPDomain$
  if $token \not\equiv null$ **then**
    let $n_1 = \texttt{RANDOM}()$
    let $m' := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, s'.IdP.Host, s'.IdP.UIDToken, \langle\langle \texttt{UIDToken}, token \rangle\rangle\rangle$
    stop $\langle s'.IdP.Address, \texttt{\_SELF}, m' \rangle, s'$
let $n, headers, body$ **such that** $\langle \texttt{HTTPResp}, n, 200, headers, body \rangle \equiv m$
  if *possible*; **otherwise** stop $\langle \rangle, s'$
if $n \equiv n_1$ **then**
  let $uid := body[UID]$
  if $uid \not\equiv null$ **then**
    let $key := \texttt{GenerateKey}()$
    let $PRPID := \texttt{Encrypt}(s'.Parameters[RPDomain], key)$
    let $PPID := \texttt{Encrypt}(\texttt{Hash}(s'.Parameters[RPDomain], uid), key)$
    let $n_2 = \texttt{RANDOM}()$
    let $m' := \langle \texttt{HTTPReq}, n_2, \texttt{POST}, s'.IdP.Host, s'.IdP.IdentityToken, \langle\langle \texttt{PRPID}, PRPID \rangle, \langle \texttt{PPID}, PPID \rangle\rangle\rangle$
    stop $\langle s'.IdP.Address, \texttt{\_SELF}, m' \rangle, s'$
else if $n \equiv n_2$ **then**
  let $token := body[Token]$
  let $m' := \langle \texttt{ENCLAVEMESSAGE}, \langle\langle \texttt{Token}, token \rangle, \langle \texttt{Key}, key \rangle\rangle\rangle$
  stop $\langle \texttt{InvokeFrom}, \texttt{\_SELF}, m' \rangle, s'$
stop $\langle \rangle, s'$