Enhancing the User Privacy of SSO by Integrating PPID and SGX

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, identified by NIST. That is (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. In this specification, the Pairwise Pseudonymous Identifier (PPID) is suggested for SSO systems to prevent collusive RPs from linking the same user. PPID mechanism enables an IdP to provide a user multiple individual IDs for different RPs. And PPID mechanism is also provided by popular SSO protocols, such as OpenID Connect and SAML. It also suggested by some identity service providers such as NORDIC APIS and CURITY. But PPID mechanism cannot protect user from IdP's tracing, as the generation of PPID requires the participation of RP ID .
In this paper, we propose an SSO system, Enhancing the User Privacy of SSO by Integrating PPID and SGX, also named *UP-SSO* . UP-SSO provides the enhanced PPID mechanism 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 generation of PPID is shifted from IdP server to user client, so that IdP server no longer needs to learn RP ID. Moreover, the user client is protected by Intel SGX, therefore, it can verify RP ID, generate PPID and transmit it to IdP server honestly. It also transforms RP ID into one-time encrypted ID, used by IdP for identity token generation without learning RP's real ID. The correctness of client can be verified by IdP by remote attestation. The server-part service running on the IdP server still takes the responsibility of authenticating users, storing the private key, issuing the identity token and etc. The detailed design of UP-SSO is described in this paper, and the systemically analysis is provided to guarantee its security. We implemented the prototype system of UP-SSO, and the evaluation of prototype system shows the overhead is modest.

Keywords: SSO · Privacy · Intel SGX.

1 Introduction

Single sign-on (SSO) systems, such as OAuth, OpenID Connect and SAML, have been widely adopted nowadays as a convenient web authentication mechanism.

SSO delegates user authentication on websites to a third party, so-called identity providers (IdPs), so that users can access different services on cooperating sites, so-called relying parties (RPs), via a single authentication process. Using SSO, a user no longer needs to maintain multiple credentials for different RPs, instead, she only maintains the credential for the IdP, which further provides identity proofs to RPs. For RPs, SSO shifts the burden of user authentication to IdPs and therefore reduces their own security risks and costs. As a result, SSO has been widely integrated with modern web systems. According to Alexa [**?**], 80 websites among the top-100 websites integrate SSO services. Meanwhile, many email and social networking providers such as Google, Facebook, Twitter, etc. have been actively serving as social identity providers to support social login.

However, the wide adoption of SSO also raises new privacy concerns. NIST [**?**] indicates that 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.

The IdP-based login tracing and RP-based identity linkage would bring the real threat in reality. Imagine that, a user concerning her privacy would avoid to leave her full sensitive information at an application. She only leaves parts of her sensitive messages at each applications, for example, using real name on a social website and address on shopping website. And she would try not to leave any linkable message to avoid applications combining her informations, such as 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 her information based on the SSO account.

To protect user privacy, the Pairwise Pseudonymous Identifier (PPID) is suggested by NIST [**?**] and adopted by some popular SSO protocols, such as OpenID Connect [**?**, **?**] and SAML [**?**]. In the PPID SSO system, IdP offers a user multiple individual IDs for different RPs. Therefore, an RP cannot correlate the received PPID with the user's PPID at another RP. Thus, collusive RPs cannot link a user's logins from her PPIDs. Recently more and more IdPs have adopted PPID to protect user privacy. Moreover, Active Directory Federation Services and Oracle Access Management support the use of PPID, the privacy protection scheme suggested in OIDC protocol [**?**, **?**]. Identity service providers such as NORDIC APIS and CURITY suggest adopting PPID in SSO to protect user privacy [**?**, **?**].

There have already been many works on protecting user privacy in SSO systems. However, to the best of our knowledge, none of the existing techniques can be integrated into the existing PPID systems. Here we give a brief introduction of existing solutions for privacy-preserving SSO and explain the flaws of these schemes.

- *Simply hiding RP ID from IdP.* For example, SPRESSO [**?**] 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, PPID is not available in this type of solution, as IdP cannot offer an RP the constant user ID with one-time irrelevant RP ID.
- *Proving user identity based on zero-knowledge proof.* In this type of solution , such as EL PASSO [**?**] and UnlimitID [**?**], 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 [**?**], 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.

We remark that the problems cannot be solved, such as simply combining existing solutions together. The challenge is that, there is not a simple way for IdP to provide PPID without knowing the RP's identity.

In this paper, we propose Enhancing the User Privacy of SSO by Integrating PPID and SGX (named UP-SSO), the enhanced PPID SSO system with comprehensive protection against bosh IdP-based login tracing and RP-based identity linkage. The key idea of UP-SSO is shifting part of IdP service, PPID generation, from server to user client. The user-side IdP service takes the responsibility for transforming RP ID (as an identity token must contain the RP ID to avoid misuse of token [**?**, **?**, **?**, **?**, **?**]) and generating PPID. For IdP, it can only achieve a encrypted one-time RP ID, so that the IdP-based login tracing is not impossible. To be noticed is that the user-side service must be deployed at the user controlled and IdP trusted environment, i.e., Intel SGX. With SGX, the secure hardware supported by intel CPU, the user-side service can be deployed at user's PC, and it can be verified by IdP through *remoteattestation*, the function provided by SGX. Moreover, other essential IdP services, such as user authentication and identity proof issuing, must be completed at server side for security consideration.

The overview of UP-SSO is that

We summarize our contributions as follows.

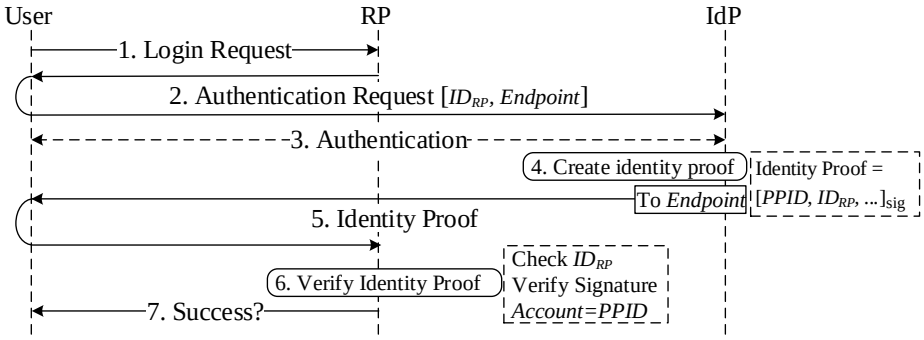

Fig. 1: The implicit protocol flow of OIDC.

- 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 (e.g., OIDC), and demonstrate its efficiency.

The rest of the paper is organized as follows. We first introduce the background in Section 2. Then, we describe the threat model, and our XXX design in Sections 3 and 4, followed by a formal analysis of security and privacy in Section 5. We provide the implementation specifics and experiment evaluation in Section 6, discuss the related works in Section 7, and conclude our work in Section 8.

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 [**?**] is an extension of OAuth 2.0 to support user authentication, and has become 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 choose 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.

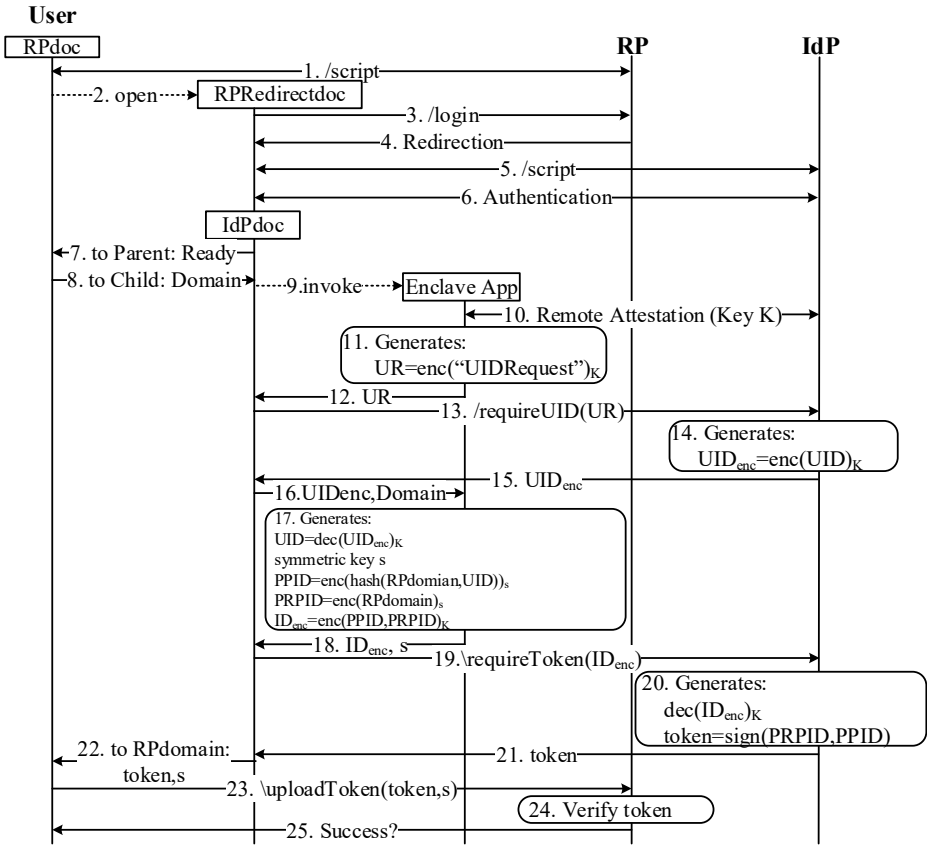

Fig. 2: The protocol flow of XXX.

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 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 IdP script invokes the enclave application (step 9). The enclave application conducts remote attestation at its initialized execution (step 10). After the remote attestation, enclave application and IdP server shares a symmetric key K.

The enclave application generates the UID request (i.e., UR) by encrypted request information with K (step 11), and return it back to IdP script (step 12). Then the user starts the UID request to IdP (step 13). IdP encrypts UID with K (step 14) and sends it to user (step 15).

While IdP script receives the encrypted UID, it transmits it to enclave application with RP's domain (step 16). The enclave application derives the UID with K, generates the symmetric key s, encrypts the RP's domain with key s as the transformed RP ID (i.e., $PRPID$), encrypts the hash of RP's domain and UID as the $PPID$, and encrypts $PRPID$ and $PPID$ with K (step 17). After the ID transformation, enclave application sends the encrypted IDs and s to IdP script (step 18).

Then user sends the encrypted $PRPID$ and $PPID$ to IdP for identity token (19). The IdP server derives the $PRPID$ and $PPID$, signs the *token* consisted of $PRPID$ and $PPID$ as the identity proof (step 20), and returns it to enclave application (step 21). The IdP script then sends the *token* and s to the origin $RPdomain$ through $postMessage$ (step 22). 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 23). 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 24). Then it returns the login result to user (step 25).

5 Security Analysis

Our formal analysis of XXX is based on the Dolev-Yao style web model [**?**], which has been widely used in formal analysis of SSO protocol, e.g., OAuth 2.0 [**?**] and OIDC [**?**]. 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 [**?**].

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 public key of the IdP. 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 satisfy 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. We first give an outline 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. We adopt SHA-256 for digest generation, RSA-2048 for signature generation, and 128-bit AES/GCM algorithm for symmetrical encryption.

IdP server and RP server. IdP and RP servers are implemented based on Spring Boot, a popular web framework for the Java platform. We also use the open-source auth0/java-jwt library to generate and verify the identity token (in

the form of Json Web Token). The encryption, decryption and other cryptographic computations are implemented using API provided by Java SDK. Moreover, the servers offer user interfaces to download JavaScript codes.

Native Messaging. Beside of parties required in XXX system, we adopt chrome extension to enable the JavaScript code to invoke the enclave application. The chrome extension invokes a native application through the Native Messaging mechanism. It requires a user to update her Windows Registry by running a .reg profile. The profile defines the location of native application and which extension can invoke the application (identified by chrome-extension ID).

Enclave application. Intel provides the enclave SDK for developers to develop enclave applications for C++ platform. Enclave SDK includes the APIs for cryptographic computations, under hardware-level protection. An enclave application contains two parts, the inside and outside SGX parts. In our implementation, the outside part only takes responsibility of transmitting data between JavaScript code and inside-SGX-part enclave application.

6.2 Evaluation

Environment. In the evaluation, we deploy the RP and IdP servers on the device with Intel i7-8700 CPU and 32GB RAM memory, running Windows 10 x64 system. The browser is Chrome v90.0.4430.212, deployed on the same device.

Result. We have run SSO process on our prototype system 100 times, and the average time is 154 ms (excluding the overhead of remote attestation). For comparison, we run MITREid Connect, a popular open-source OpenID Connect implementation in Java. The time cost of MITREid Connect is 113 ms. The overhead is modest.

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 [**?**,**?**]. Some other attackers try to temper the identity token to impersonate an honest user, such as XML Signature wrapping (XSW) [**?**], RP's incomplete verification [**?**,**?**,**?**], IdP spoofing [**?**,**?**] 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 [**?**,**?**,**?**,**?**,**?**]. Moreover, automatical tools, such as SSOScan [**?**], OAuthTester [**?**] and S3KVetter [**?**], 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. [**?**, **?**] analyse the OAuth 2.0 and OIDC protocols based on the expressive Dolev-Yao style model [**?**]. 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 [**?**] 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 [**?**] and OIDC [**?**]. Some protocols simply hide the RP's identity from IdP, such as SPRESSO [**?**] (using encrypted RP identifier) and BrowserID [**?**] (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 [**?**, **?**]. 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 [**?**] and UnlimitID [**?**]. 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 [**?**, **?**]. 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 overhead is modest.

References

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 $\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle$. We assume there is an HTTP request $r := \langle \texttt{HTTPReq}, n, \texttt{GET}, example.com, /path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$, here we define the operation on the r. That is, the elements in r can be accessed in the form $r.name$, such that $r.method \equiv \texttt{GET}$, $r.path \equiv /path$ 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 $\langle \texttt{IFRAME}, URL, WindowNonce\rangle$ 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: $\langle a, f, m\rangle, s$

```
1: let s' := s
2: let n, method, path, parameters, headers, body such that
     ⟨HTTPReq, n, method, path, parameters, headers, body⟩ ≡ m
     if possible; otherwise stop ⟨⟩, s'
3: if path ≡ /script then
4:     let m' := ⟨HTTPResp, n, 200, ⟨⟩, IdPScript⟩
5:     stop ⟨f, a, m'⟩, s'
6: else if path ≡ /login then
7:     let cookie := headers[Cookie]
8:     let session := s'.sessions[cookie]
9:     let username := body[username]
10:    let password := body[password]
11:    if password ≢ SecretOfID(username) then
12:        let m' := ⟨HTTPResp, n, 200, ⟨⟩, LoginFailure⟩
13:        stop ⟨f, a, m'⟩, s'
14:    let session[uid] := UIDOfUser(username)
15:    let m' := ⟨HTTPResp, n, 200, ⟨⟩, LoginSucess⟩
16:    stop ⟨f, a, m'⟩, s'
17: else if path ≡ /requireUID then
18:    let cookie := headers[Cookie]
19:    let request := body[request]
20:    let session := s'.sessions[cookie]
21:    let uid := session[uid]
22:    let key := session[key]
23:    if uid ≢ null&Decrypt(request) ≡ UIDRequest then
24:        let UID_enc := Encrypt(uid, key)
25:        let m' := ⟨HTTPResp, n, 200, ⟨⟩, ⟨UID, UID_enc⟩⟩
26:        stop ⟨f, a, m'⟩, s'
27: else if path ≡ /requireToken then
28:    let cookie := headers[Cookie]
29:    let IDs := body[IDs]
30:    let session := s'.sessions[cookie]
31:    let key := session[key]
32:    let decIDs := Decrypt(IDs, key)
33:    let PRPID := decIDs.PRPID
34:    let PPID := decIDs.PPID
35:    let Content := ⟨PRPID, PPID⟩
36:    let token := Content + Sign(Content, s'.SignKey)
37:    let m' := ⟨HTTPResp, n, 200, ⟨⟩, ⟨Token, token⟩⟩
38:    stop ⟨f, a, m'⟩, 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**
$\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$
if *possible*; **otherwise** stop $\langle\rangle, s'$
3: **if** $path \equiv /script$ **then**
4: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript}\rangle$
5: **stop** $\langle f, a, m'\rangle, s'$
6: **else if** $path \equiv /uploadToken$ **then**
7: **let** $cookie := headers[Cookie]$
8: **let** $session := s'.sessions[cookie]$
9: **let** $token := body[Token]$
10: **let** $key := body[Key]$
11: **if** $\texttt{CheckSig}(Token.Content, Token.Sig, s'.IdP.PubKey)$ **then**
12: **if** $\texttt{Encrypt}(s'.RPDomain, key) \equiv token.Content.PRPID$ **then**
13: **let** $PPID := \texttt{Decrypt}(token.Content.PPID, key)$
14: **if** $PPID \in \texttt{ListOfUser}()$ **then**
15: **let** $session[user] := PPID$
16: **let** $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSuccess}\rangle$
17: **stop** $\langle f, a, m'\rangle, s'$

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** $target := \texttt{PARENTWINDOW}(tree, docnonce)$
3: **let** $IdPDomain := s'.IdPDomain$
4: **switch** $s'.q$ **do**
5: **case** *startLogin*:
6: **let** $username \in ids$
7: **let** $Url := \langle \texttt{URL}, \texttt{S}, IdPDomain, /login, \langle\rangle\rangle$
8: **let** $command := \langle \texttt{XMLHTTPREQUEST}, Url, \texttt{POST},$
$\langle\langle username, username\rangle, \langle password, secret\rangle\rangle, s'.refXHR\rangle$
9: **let** $s'.q := expectLoginResult$

```
10:    case expectLoginResult:
11:        let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
12:        let input := CHOOSEINPUT(scriptinputs, pattern)
13:        if input ≠ null&input.Body ≡ LoginSuccess then
14:            let command := ⟨POSTMESSAGE, target, Ready, null⟩
15:            let s′.q := expectRPDomain
16:    case expectRPDomain:
17:        let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
18:        let input := CHOOSEINPUT(scriptinputs, pattern)
19:        if input ≠ null then
20:            let s′.Parameters[RPDomain] := input.Content[RPDomain]
21:            let Url := ⟨URL, S, IdPDomain, /requireUIDToken, ⟨⟩⟩
22:            let command := ⟨ENCLAVEMESSAGE, ⟨⟨Type, UIDRequest⟩⟩⟩
23:            let s′.q := expectUIDRequest
24:    case expectUIDRequest:
25:        let pattern := ⟨ENCLAVEMESSAGE, Content⟩
26:        let input := CHOOSEINPUT(scriptinputs, pattern)
27:        if input ≠ null&input.Content[Type] ≡ UIDRequest then
28:            let UIDRequest := input.Content[UIDRequest]
29:            let command := ⟨XMLHTTPREQUEST, Url, POST,
               ⟨⟨UIDRequest, UIDRequest⟩⟩, s′.refXHR⟩
30:            let s′.q := expectUID
31:    case expectUID:
32:        let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
33:        let input := CHOOSEINPUT(scriptinputs, pattern)
34:        if input ≠ null then
35:            let UID := input.Body[UID_enc]
36:            let command := ⟨ENCLAVEMESSAGE, ⟨⟨Type, UID⟩
               ⟨UID, UID⟩, ⟨Domain, s′.Parameters[RPDomain]⟩⟩
37:            let s′.q := expectIDs
38:    case expectIDs:
39:        let pattern := ⟨ENCLAVEMESSAGE, Content⟩
40:        let input := CHOOSEINPUT(scriptinputs, pattern)
41:        if input ≠ null&input.Content[Type] ≡ IDs then
42:            let IDs := input.Content[IDs]
43:            let s′.Parameters[Key] := input.Content[Key]
44:            let command := ⟨XMLHTTPREQUEST, Url, POST,
               ⟨⟨IDs, IDs⟩⟩, s′.refXHR⟩
45:            let s′.q := expectIdentityToken
46:    case expectIdentityToken:
47:        let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
48:        let input := CHOOSEINPUT(scriptinputs, pattern)
49:        if input ≠ null then
50:            let token := Body[Token]
51:            let command := ⟨POSTMESSAGE, target, ⟨⟨IdentityToken, token⟩,
```

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

Enclave application. The state of enclave application is a term in the form $\langle Parameters, Key \rangle$.
- $Parameters$ is used to store the parameters received from other processes.
- Key is the key shared with IdP.

Algorithm 5 R^e

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

```
1: let s' := s
2: if m.Content[type] ≡ UIDRequest then
3:     let UIDRequest := Encrypt(UIDRequest, s'.Key)
4:     let m' := ⟨⟨Type, UIDRequest⟩, ⟨UIDRequest, UIDRequest⟩⟩
5:     stop ⟨f, a, m'⟩, s'
6: else if m.Content[type] ≡ UID then
7:     let Domain := m.Content[Domain]
8:     let UID_enc := m.Content[UID]
9:     let UID := Decrypt(UID_enc, s'.Key)
10:    let Key' := GenerateKey()
11:    let PRPID := Encrypt(Domain, Key')
12:    let PPID := Encrypt(Hash(Domain, UID), Key')
13:    let IDs := Encrypt(PRPID + PPID, s'.Key)
14:    let m' := ⟨⟨Type, IDs⟩, ⟨IDs, IDs⟩, ⟨Key, Key'⟩⟩
15:    stop ⟨f, a, m'⟩, s'
```