Contribution Title*

First Author[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Single sign-on (SSO) systems, such as OpenID Connect [**?**], OAuth [**?**] and SAML [**?**], 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 [**?**] support SSO, and the analysis on the Alexa Top-1M websites [**?**] 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 [**?**] and Onavo [**?**]). 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

* Supported by organization x.

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 [**?**, **?**]; identity service providers such as NORDIC APIS and CURITY suggest adopting PPID in SSO to protect user privacy [**?**, **?**].

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 [**?**] and specified in several SSO protocols [**?**, **?**], 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 [**?**] 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, 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.

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 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 [**?**] is an extension of OAuth 2.0 to support user authentication, and becomes one of the most prominent SSO authentication protocols. Same as other SSO protocols [**?**], OIDC involves three entities, i.e., *users*, *identity provider (IdP)*, and *relying parties (RPs)*. Both users and RPs have to register at the IdP, the users register at the IdP to create credentials and identifiers (i.e. ID_U), while each RP registers at the IdP with its endpoint information to create its unique identifier (i.e., ID_{RP}) and the corresponding credential. IdP is assumed to securely maintain the attributes of users and RPs. Then, in the SSO authentication sessions, each user is responsible to start a login request at an RP, redirect the messages between RP and IdP, and check the scope of user's attributes provided to the RP; IdP authenticates the user, sets the $PPID$ for the user ID_U at the RP ID_{RP}, constructs the identity proof with $PPID$, ID_{RP} and the user's attributes consented by the user, and finally transmits the identity proof to the RP's registered endpoint (e.g., URL); each RP constructs an identity proof request with its identifier and the requested scope of user's attributes, sends an identity proof request to the IdP through the user, and parses the received identity proof to authenticate and authorize the user. Usually, the redirection and checking at the user are handled by a user-controlled software, called *user agent* (e.g., browser).

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).

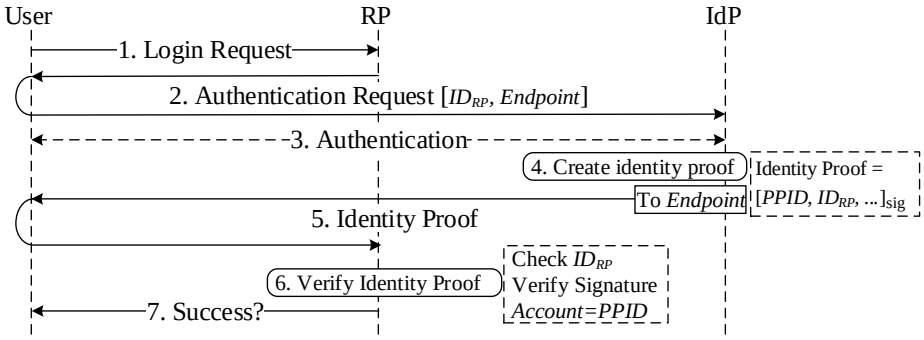

Fig. 1: The implicit protocol flow of OIDC.

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' Goals: (1) The adversaries can impersonate an honest user to log in to the honest RP. (2) The adversary lead the honest user to send the malicious identity proof (generated by the adversary) 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).

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.

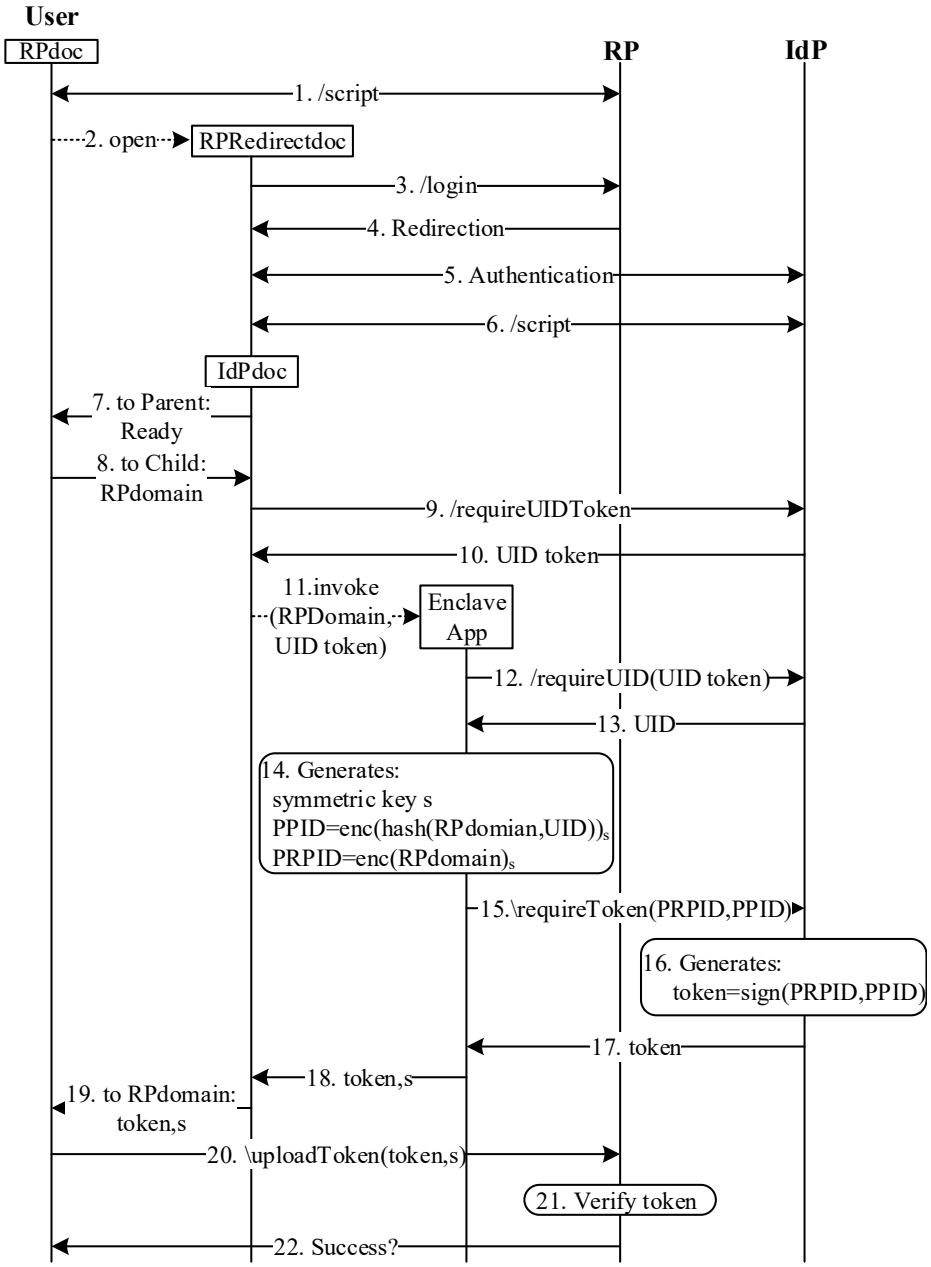

Fig. 2: The protocol flow of XXX.

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 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 [**?**], 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'.

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 UPPRESSO 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 Proof of Security

A Appendix: Web Model

A.1 Message Format

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

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

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

An HTTP response message is the term of the form

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

The details are defined as follows:

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

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

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

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

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

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 $body[password] \equiv 123$.

A.2 Browser Model

In UPPRESSO, we assume that the browsers are honest, therefore, we only need to analyze how the browsers interactive with the scripts. We first introduce the windows and documents of the browser model.

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

Document. A document d is a term of the form

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

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

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

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

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

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

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

The *tree* is the relations of the opened windows and documents, which are visible to this script. *Docnonce* is the document nonce. The *Scriptstate* is a term of the form defined by each script. *Scriptinputs* is the message transmitted to script. However, the *scriptinputs* is defined as standardized forms, for example, postMessage is one of the forms of *scriptinputs*. *Cookies* is the set of cookies that belong to the document's origin. *LocalStorage* is the storage space for browser and *sessionStorage* is the space for each HTTP sessions. *Ids* is the set of user IDs while *secret* is the password to corresponding user ID. The *command* is the operation which is to be conducted by the browser. Here we only introduce the form of commands used in 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.3 XXX Model

IdP server.

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 $\langle f, a, m'\rangle, s'$
    end if
    let $session[uid] := \texttt{UIDOfUser}(username)$
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess}\rangle$
    stop $\langle f, a, m'\rangle, s'$

20: **else if** $path \equiv /requireUIDToken$ **then**
21: **let** $cookie := headers[Cookie]$
22: **let** $session := s'.sessions[cookie]$
23: **let** $uid := session[uid]$
24: **if** $uid \equiv$ `null` **then**
25: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle,$ `UnLogged`$\rangle$
26: **stop** $\langle f, a, m'\rangle, s'$
27: **end if**
28: **let** $token :=$ `GenerateToken`()
29: **let** $s'.Tokens := s'.Tokens +^{\langle\rangle} \langle uid, token\rangle$
30: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle, \langle$`Token`$, token\rangle\rangle$
31: **stop** $\langle f, a, m'\rangle, s'$
32: **else if** $path \equiv /requireUID$ **then**
33: **let** $UIDToken := body[UIDToken]$
34: **let** $uid :=$ `FindUIDByToken`(`UIDToken`)
35: **if** $uid \equiv null$ **then**
36: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle,$ `TokenError`$\rangle$
37: **stop** $\langle f, a, m'\rangle, s'$
38: **end if**
39: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle, \langle$`UID`$, uid\rangle\rangle$
40: **stop** $\langle f, a, m'\rangle, s'$
41: **else if** $path \equiv /requireToken$ **then**
42: **let** $PRPID := body[PRPID]$
43: **let** $PPID := body[PPID]$
44: **let** $Content := \langle PRPID, PPID\rangle$
45: **let** $token := Content + Sign(Content, s'.SignKey)$
46: **let** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle, \langle$`Token`$, token\rangle\rangle$
47: **stop** $\langle f, a, m'\rangle, s'$
48: **end if**
49: **stop** $\langle\rangle, s'$

RP server.

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: $\langle$`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** $m' := \langle$`HTTPResp`$, n, 200, \langle\rangle,$ `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** $token := body[Token]$

```
    let key := body[Key]
    if CheckSig(Token.Content, Token.Sig, s'.IdP.PubKey) then
        let PRPID := Encrypt(s'.RPDomain, key)
        if PRPID ≡ token.Content.PRPID then
            let PPID := token.Content.PPID
            if PPID ∉ ListOfUser() then
                let RegisterUser(PPID)
            end if
            let session[user] := PPID
            let m' := ⟨HTTPResp, n, 200, ⟨⟩, LoginSuccess⟩
        end if
    end if
    let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
    stop ⟨f, a, m'⟩, s'
end if
stop ⟨⟩, s'
```

IdP script.

Algorithm 3 *script_idp*

```
Input: ⟨tree, docnonce, scriptstate, scriptinputs, cookies, localStorage,
    sessionStorage, ids, secret⟩
let s' := scriptstate
let command := ⟨⟩
let target := PARENTWINDOW(tree, docnonce)
let IdPDomain := s'.IdPDomain
switch s'.q do
    case startLogin:
        let username ∈ ids
        let Url := ⟨URL, S, IdPDomain, /login, ⟨⟩⟩
        let s'.refXHR := Random()
        let command := ⟨XMLHTTPREQUEST, Url, POST,
            ⟨⟨username, username⟩, ⟨password, secret⟩⟩, s'.refXHR⟩
        let s'.q := expectLoginResult
    end case
    case expectLoginResult:
        let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            if input.Body ≢ LoginSuccess then
                let stop ⟨⟩
            end if
            let command := ⟨POSTMESSAGE, target, Ready, null⟩
            let s'.q := expectRPDomain
        end if
    end case
```

```
    case expectRPDomain:
        let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≠ null then
            let RPDomain := input.Content[RPDomain]
            let s′.Parameters[RPDomain] := RPDomain
            let Url := ⟨URL, S, IdPDomain, /requireUIDToken, ⟨⟩⟩
            let s′.refXHR := Random()
            let command := ⟨XMLHTTPREQUEST, Url, GET, s′.refXHR⟩
            let s′.q := expectUIDToken
        end if
    end case
    case expectUIDToken:
        let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≠ null then
            let token := input.Body[UIDToken]
            let command := ⟨ENCLAVEMESSAGE, ⟨UIDToken, token⟩⟩
            let s′.q := expectIdentityToken
        end if
    end case
    case expectIdentityToken:
        let pattern := ⟨ENCLAVEMESSAGE, Content⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≠ null then
            let token := input.Content[Token]
            let key := input.Content[Key]
            let command := ⟨POSTMESSAGE, target, ⟨⟨IdentityToken, token⟩,
              ⟨Key, key⟩⟩, s′.Parameters[RPDomain]⟩
            let s′.q := stop
        end if
    end case
end switch
let stop ⟨s′, cookies, localStorage, sessionStorage, command⟩
```

RP script.

Algorithm 4 *script_rp*

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secret*⟩

```
let s′ := scriptstate
let command := ⟨⟩
let IdPWindow := SUBWINDOW(tree, docnonce).nonce
let RPDomain := s′.RPDomain
switch s′.q do
    case start:
```

```
7:        let Url := ⟨URL, S, RPDomain, /login, ⟨⟩⟩
8:        let command := ⟨IFRAME, Url, _SELF⟩
9:        let s'.q := expectReady
10:   end case
11:   case expectReady:
12:       let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
13:       let input := CHOOSEINPUT(scriptinputs, pattern)
14:       if input ≢ null ∧ input.Content ≡ Ready then
15:           let Content := ⟨RPDomain, RPDomain⟩
16:           let command := ⟨POSTMESSAGE, target, Content, null⟩
17:           let s'.q := expectToken
18:       end if
19:   end case
20:   case expectToken:
21:       let pattern := ⟨POSTMESSAGE, ∗, Content, ∗⟩
22:       let input := CHOOSEINPUT(scriptinputs, pattern)
23:       if input ≢ null then
24:           let token := input.Content[Token]
25:           let key := input.Content[Token]
26:           let Url := ⟨URL, S, RPDomain, /uploadToken, ⟨⟩⟩
27:           let s'.refXHR := Random()
28:           let command := ⟨XMLHTTPREQUEST, Url, POST,
              ⟨⟨Token, token⟩, ⟨Key, key⟩⟩, s'.refXHR⟩
29:           let s'.q := stop
30:       end if
31:   end case
32: end switch
```

Enclave application.

Algorithm 5 R^e

```
Input: ⟨a, f, m⟩, s
1: let s' := s
2: let InvokeFrom := f
3: if m.type ≡ ENCLAVEMESSAGE then
4:     let token := m.Content[Token]
5:     let RPDomain := m.Content[RPDomain]
6:     let s'.Parameters[RPDomain] := RPDomain
7:     if token ≢ null then
8:         let n1 = RANDOM()
9:         let m' := ⟨HTTPReq, n1, POST, s'.IdP.Host, s'.IdP.UIDToken,
           ⟨⟨UIDToken, token⟩⟩⟩
10:        stop ⟨s'.IdP.Address, _SELF, m'⟩, s'
11:    end if
12: end if
13: let n, headers, body such that ⟨HTTPResp, n, 200, headers, body⟩ ≡ m
```

14: **if** *possible*; **otherwise** stop $\langle\rangle, s'$
15: **if** $n \equiv n_1$ **then**
16: **let** $uid := body[UID]$
17: **if** $uid \not\equiv null$ **then**
18: **let** $key :=$ `GenerateKey`()
19: **let** $PRPID :=$ `Encrypt`$(s'.Parameters[RPDomain], key)$
20: **let** $PPID :=$ `Encrypt`(`Hash`$(s'.Parameters[RPDomain], uid), key)$
21: **let** $n_2 =$ `RANDOM`()
22: **let** $m' := \langle$`HTTPReq`$, n_2,$ `POST`$, s'.IdP.Host, s'.IdP.IdentityToken,$
$\langle\langle$`PRPID`$, PRPID\rangle, \langle$`PPID`$, PPID\rangle\rangle\rangle$
23: **stop** $\langle s'.IdP.Address,$ `_SELF`$, m'\rangle, s'$
24: **end if**
25: **else if** $n \equiv n_2$ **then**
26: **let** $token := body[Token]$
27: **let** $m' := \langle$`ENCLAVEMESSAGE`$, \langle\langle$`Token`$, token\rangle, \langle$`Key`$, key\rangle\rangle\rangle$
28: **stop** $\langle$`InvokeFrom`$,$ `_SELF`$, m'\rangle, s'$
29: **end if**
30: **stop** $\langle\rangle, s'$