A Preparation

Our formal security analysis of UPPRESSO is based on the general Dolev-Yao web model in SPRESSO. To facilitate the definition of UPPRESSO, we change some details in model. In particular, we add some extra function symbols for asymmetric encryption/decryption.

Since our model is using ECC(Elliptic Curve Cryptography) to encrypt/decrypt the data, we add the following symbols to the signature Σ for the terms and messages:

- $\mathbb{E}$ is an elliptic curve over a finite field $\mathbb{F}_q$, G is a base point(or generator) of $\mathbb{E}$ and the order of G is a prime number n.
- $[t]P$ means using asymmetric key t to encrypt the point $P = [p]G$ on the elliptic curve where p is the actual plaintext.
- $[t^{-1}]C$ means using the reverse of t to decrypt the point $C = [c]G = [tm]G$ on the elliptic curve where c is the cipertext.
- $\texttt{isValid}(P)$ checks whether P is a valid point on the elliptic curve. That is to say whether $P = [m]G$ for the base point G and some nonce m.

B Formal Model of UPPRESSO

We here present the full details of our formal model of UPPRESSO. For our analysis regarding our authentication and privacy properties below, we will further restrict this generic model to suit the setting of respective analysis.

We model UPPRESSO as a web system. We call a web system $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an UPPRESSO web system if it is of the form described in what follows.

B.1 Outline

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$ consists of web attacker processes (in Web), network attacker processes (in Net), a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set IDP of web servers for the identity providers, and a finite set DNS of DNS servers, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP} \cup \mathsf{DNS}$. More details on the processes in $\mathcal{W}$ are provided below. Figure 1 shows the set of scripts $\mathcal{S}$ and their respectice string representations that are defined by the mapping script. The set E^0 contains only the trigger events.

This outlines $\mathcal{UWS}$. We will define the DY processes in $\mathcal{UWS}$ and their addresses, domain names, and secrets in more detail. The scripts are defined in detail in Appendix B.14

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
$script\_rp$	`script_rp`
$script\_idp$	`script_idp`

Figure 1: List of scripts in $\mathcal{S}$ and their respective string representations.

B.2 Addresses and Domain Names

The set IPs contains for every web attacker in Web, every network attacker in Net, every relying party in RP, every identity provider in IDP, every DNS server in DNS, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every relying party in RP, every identity provider in IDP, every web attacker in Web, and every network attacker in Net. Browsers (in B) and DNS servers (in DNS) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

B.3 Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into four sets, an infinite sequence N, an infinite set K_{SSL}, an infinite set K_{sign}, an infinite set K_{id} and a finite set $\mathsf{Secrets}$. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{SSL}}}_{\text{finite}} \dot\cup \underbrace{K_{\text{sign}}}_{\text{finite}} \dot\cup \underbrace{K_{\text{id}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Secrets}}_{\text{finite}} .$$

The set N contains the nonces that are available for each DY process in $\mathcal{W}$ (it can be used to create a run of $\mathcal{W}$).

The set K_{SSL} contains the keys that will be used for SSL encryption. Let $\mathsf{sslkey}\colon \mathsf{Doms} \to K_{\text{SSL}}$ be an injective mapping that assigns a (different) private key to every domain.

The set K_{sign} contains the keys that will be used by IdPs for signing IAs. Let $\mathsf{signkey}\colon \mathsf{IdPs} \to K_{\text{sign}}$ be an injective mapping that assigns a (different) private key to every identity provider.

The set K_{id} contains all numbers $x \in [1, n)$ in which n is a prime number up to 2^{256} The set K_{id} will be used to generate identities of B and RP.

The set $\mathsf{Secrets}$ is the set of passwords (secrets) the browsers share with the identity providers.

B.4 Identities

Identities are alike email addresses, which consist of a number, a user name and a domain part. For our model, this is defined as follows:

Definition 1. *An* identity u *is a term of the form* $\langle id, name, domain\rangle$ *with* $id \in K_{id}$*,* $name \in \mathbb{S}$ *and* $domain \in \mathsf{Doms}$*.*

Let ID *be the finite set of identities. By* ID^y *we denote the set* $\{\langle id, name, domain\rangle \in \mathsf{ID} \mid domain \in \mathsf{dom}(y)\}$*.*

We say that an ID is governed *by the DY process to which the domain of the ID belongs. Formally, we define the mapping* $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$*,* $\langle id, name, domain\rangle \mapsto \mathsf{dom}^{-1}(domain)$*.*

The governor of an ID will usually be an IdP, but could also be the attacker.

By $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Secrets}$ we denote the bijective mapping that assigns secrets to all identities.

Let $\mathsf{ownerOfSecret} : \mathsf{Secrets} \to \mathsf{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret. Now, we define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$, $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).

It should be pointed out that in UPPRESSO, the relying parties also have identities referred to as r which is important for privacy analysis. The form of r is the same as u. To be concise, we usually use u and r to refer to $u.id$ and $r.id$ if we don't say they are identities or $u, r \in \mathsf{ID}$

B.5 Tags, Identity Tokens and Service Tokens

Definition 2. *A* tag *is a term of the form* $PID_{rp} = [t]ID_{rp} = [tr]G$ *for a nonce (here used as a asymmetric key)* t*.*

Definition 3. *An* identity Tokens (IDToken) *is a term of the form* $\langle PID_{rp}, PID_u, ver\rangle$ *for a tag* PID_{rp}*, an encrypted identity* $PID_u = [u]PID_{rp} = [utr]G$ *and a signature* $ver = \mathsf{sig}(\langle PID_{rp}, PID_u\rangle, k)$ *for a nonce* k*.*

Definition 4. *A* service token *is a term of the form* $Acct = [t^{-1}]PID_u = [t^{-1}][utr]G = [ur]G$ *for a nonce* $t \in K_{id}$*.*

B.6 Corruption

RPs and IdPs can become corrupted: If they receive the message `CORRUPT`, they start collecting all incoming messages in their state and (upon triggering) send out all messages that are derivable from their state and collected input messages, just like the attacker process. We say that an RP or an IdP is *honest* if the according part of their state (s.`corrupt`) is $\perp$, and that they are corrupted otherwise.

We are now ready to define the processes in $\mathcal{W}$ as well as the scripts in $\mathcal{S}$ in more detail.

B.7 Processes in $\mathcal{W}$ (Overview)

We first provide an overview of the processes in $\mathcal{W}$. All processes in $\mathcal{W}$ (except for DNS servers) contain in their initial states all public keys and the private keys of their respective domains (if any). We define $I^p = \mathsf{addr}(p)$ for all $p \in \mathsf{Hon} \cup \mathsf{Web}$.

Web Attackers. Each $wa \in \mathsf{Web}$ is a web attacker who uses only his own addresses for sending and listening.

Network Attackers. Each $na \in \mathsf{Net}$ is a network attacker who uses all addresses for sending and listening.

Browsers. Each $b \in \mathsf{B}$ is a web browser. The initial state contains all secrets owned by b, stored under the origin of the respective IdP. See Appendix B.11 for details.

Relying Parties. A relying party $r \in \mathsf{RP}$ is a web server. RP knows four distinct paths: `/script`, where it serves `script_rp` to open a new window and facilitate the login flow. `/loginSSO`, where it only accepts GET requests and sends redirect response to redirect the browser to the IdP to download `script_IdP` `/startNegotiation`, where it only accepts POST requests logically sent from `script_rp` using postMessge and checks whether the data $t \in K_{\text{id}}$. If the request valid, it send back a certificate. `/uploadToken` running in the browser. It checks the ID token and, if the data is deemed "valid", it issues a service token (again, for details, see below). Intuitively, a client having such a token can use the service of the RP (for a specific identity record along with the token). Just like IdPs, RPs can become corrupted.

Identity Providers. Each IdP is a web server, users can authenticate to the IdP with their credentials. IdP tracks the state of the users with sessions. Authenticated users can receive IDTokens from the IdP. When receiving a special message (`CORRUPT`) IdPs can become corrupted. Similar to the definition of corruption for the browser, IdPs then start sending out all messages that are derivable from their state.

DNS. Each $dns \in \mathsf{DNS}$ is a DNS server. Their state contains the allocation of domain names to IP addresses.

B.8 SSL Key Mapping

Before we define the atomic DY processes in more detail, we first define the common data structure that holds the mapping of domain names to public SSL keys: For an atomic DY process p we define

$$sslkeys^p = \langle \{ \langle d, \mathsf{sslkey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle.$$

B.9 Web Attackers

Each $wa \in \mathsf{Web}$ is a web attacker. The initial state of each wa is $s_0^{wa} = \langle attdoms, sslkeys, signkeys \rangle$, where $attdoms$ is a sequence of all domains along with the corresponding private keys owned by wa, $sslkeys$ is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all public signing keys for all IdPs. All other parties use the attacker as a DNS server.

B.10 Network Attackers

As mentioned, each network attacker na is modeled to be a network attacker. We allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle \mathit{attdoms}, \mathit{sslkeys}, \mathit{signkeys} \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker na, *sslkeys* is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all public signing keys for all IdPs.

B.11 Browsers

Each $b \in \mathsf{B}$ is a web browser with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $\mathit{ID}^b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of b, $\mathit{ID}^{b,d} := \{i \mid \exists\, x, n :\; i = \langle id, n, d\rangle \in \mathit{ID}^b\}$ be the set of IDs of b for a domain d, and $\mathit{SecretDomains}^b := \{d \mid \mathit{ID}^{b,d} \neq \emptyset\}$ be the set of all domains that b owns identities for.

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (ssl) key, according to the mapping sslkey; the DNS address is $\mathsf{addr}(p)$ with $p \in \mathcal{W}$; the list of secrets contains an entry $\langle\langle d, \mathsf{S}\rangle, s\rangle$ for each $d \in \mathit{SecretDomains}^b$ and $s = \mathsf{secretOfID}(i)$ for some $i \in \mathit{ID}^{b,d}$ (s is the same for all i); *ids* is $\langle \mathit{ID}^b \rangle$; *sts* is empty.

B.12 Relying Parties

A relying party $r \in \mathsf{RP}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$. Its initial state s_0^r contains its domains, the private keys associated with its domains and the DNS server address. The full state additionally contains the sets of service tokens and login session identifiers the RP has issued. RP only accepts HTTPS requests.

RP manages two kinds of sessions: The *login sessions*, which are only used during the login phase of a user, and the *service sessions* (we call the session identifier of a service session a *service token*). Service sessions allow a user to use RP's services. The ultimate goal of a login flow is to establish such a service session.

In a typical flow with one client, r will first receive an HTTP GET request for the path /`script`. In this case, r returns the script `script_rp` (see below).

After the user loaded the script in his browser, r will receive an HTTP GET request for the path /`loginSSO` sent from the new window opened by `script_rp`. In this request, r will send back a redirect response for downloading `script_IdP` from IdP.

When the IdP document in the browser generates a number $t \in K_{\text{id}}$, r will receive the third request for the path /`startNegotiate`. r will verify t and if valid, r will create the corresponding login session with a *loginSessionToken* as the identifier. After that, r will use t to generate PID_{rp} and bind it with the login session. After all these are down, r send its certificate signed by the specific IdP that browser selected.

Finally, r receives a last request in the login flow. This POST request contains the IDToken. To conclude the login, r looks up the user's login session, compare the $IDToken.\mathtt{PID_{rp}}$ with the PID_{rp} in the login session, and checks whether $IDToken.\mathtt{PID_{ver}}$ is a correct signature. If successful, r calculates the service token and returns it, which is also stored in the state of r.

If r receives a corrupt message, it becomes corrupt and acts like the attacker from then on.

We now provide the formal definition of r as an atomic DY process (I^r, Z^r, R^r, s_0^r). As mentioned, we define $I^r = \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 5. *A* login session record *is a term of the form* $\langle t, PID_{rp}\rangle$ *with* $t, PID_{rp} = [tr]G(t, r \in K_{id})$.

Definition 6. *A* state $s \in Z^r$ of an RP r *is a term of the form* $\langle$*DNSAddress, keyMapping, sslkeys, pendingDNS, pendingRequests, loginSessions, serviceTokens, wkCache, corrupt, IdPConfig, rp*$\rangle$ *where* $DNSAddress \in \mathsf{IPs}$, $keyMapping \in [\mathbb{S} \times \mathcal{N}]$, $sslkeys = sslkeys^r$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $serviceTokens \in \mathcal{N}$, $loginSessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ *is a dictionary of login session records,* $wkCache \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $IdPConfig \in \mathcal{T}_{\mathcal{N}}$ *is the configuration retrieved from IdP server,* $rp \in \mathsf{ID}$ *is the identity of the RP, see details in Appendix B.4.*

The initial state s_0^r of r *is a state of* r *with* $s_0^r.\mathtt{serviceTokens} = s_0^r.\mathtt{loginSessions} = s_0^r.\mathtt{wkCache} = \langle\rangle$, $s_0^r.\mathtt{corrupt} = \bot$, $s_0^r.\mathtt{keyMapping}$ *is the same as the keymapping for browsers above,* $s_0^r.\mathtt{IdPConfig} = \langle pubkey, scriptUrl, Cert_{rp}\rangle$ *and* $s_0^r.\mathtt{rp} = \langle id, name, domain\rangle$.

We now specify the relation R^r. We describe this relation by a non-deterministic algorithm.

Algorithm 1 Relation of a Relying Party R^r

```
Input: $\langle a, b, m\rangle, s$
1: let $s' := s$
2: if $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ then
3:     let $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
4:     let $m' := d_V(s')$
5:     let $a' := \mathsf{IPs}$
6:     stop $\langle a', a, m'\rangle, s'$
7: end if
8: let $m_{dec}, k, k', inDomain$ such that
    ↪ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s'.\mathtt{sslkeys}$
    ↪ if possible; otherwise stop $\langle\rangle, s'$
9: let $n, method, path, parameters, headers, body$ such that
    ↪ $\langle \mathtt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m_{dec}$
    $\hookrightarrow$ **if possible; otherwise stop** $\langle\rangle, s'$
**if** $path \equiv /script$ **then**
    **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{script\_rp}\rangle, k)$
    **stop** $\langle b, a, m'\rangle, s'$
**else if** $path \equiv /loginSSO$ **then**
    **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 302, \langle\langle \mathtt{Location}, s'.\mathtt{IdPConfig}.scriptUrl\rangle\rangle, \langle\rangle\rangle, k)$
    **stop** $\langle b, a, m'\rangle, s'$
**else if** $path \equiv /startNegotiation$ **then**
    **let** $loginSessionToken := \nu_1$
    **let** $t := body[t]$
    **let** $ID_{rp} := [s'.\mathtt{rp}.id]G$
    **let** $PID_{rp} := [t]ID_{rp}$
    **let** $state := \mathtt{expectToken}$
    **let** $s'.\mathtt{loginSessions}[loginSessionToken] := \langle t, PID_{rp}, state\rangle$
    **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \langle \mathtt{Cert_{RP}}, s'.\mathtt{IdPConfig}.Cert_{RP}\rangle\rangle, k)$
    **stop** $\langle b, a, m'\rangle, s'$
**else if** $path \equiv /uploadToken$ **then**
    **let** $loginSessions := s'.\mathtt{loginSessions}[body[\mathtt{loginSessionToken}]]$
    **if** $loginSessions \equiv \langle\rangle$ **then**
        **stop** $\langle\rangle, s'$
    **end if**
    **if** $loginSessions.\mathtt{state} \not\equiv expectToken$ **then**
        **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle, k)$
        **stop** $\langle b, a, m'\rangle, s'$
    **end if**
    **let** $s'.\mathtt{loginSessions} := s'.\mathtt{loginSessions} - body[loginSessionToken]$
    **let** $IDToken := body[\mathtt{IDToken}]$
    **if** $IDToken.\mathtt{PID_{rp}} \not\equiv loginSessions.\mathtt{PID_{rp}}$ **then**
        **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle, k)$
        **stop** $\langle b, a, m'\rangle, s'$
    **end if**
    **if** $\mathsf{checksig}(IDToken.\mathtt{ver}, \langle IDToken.\mathtt{PID_{rp}}, IDToken.\mathtt{PID_u}\rangle, s'.\mathtt{IdPConfig}.pubkey) \equiv \bot$
    **then**
        **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Fail}\rangle, k)$
        **stop** $\langle b, a, m'\rangle, s'$
    **end if**
    **let** $PID_u := IDToken.\mathtt{PID_u}$
    **let** $Acct := [loginSessions.\mathtt{t}]PID_u$
    **let** $s'.\mathtt{serviceTokens} := s'.\mathtt{serviceTokens} +^{\langle\rangle} Acct$
    **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{LoginSuccess}\rangle, k)$
    **stop** $\langle b, a, m'\rangle, s'$
**end if**
**stop** $\langle\rangle, s'$
```

B.13 Identity Providers

An identity provider $i \in \mathsf{IdPs}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s^i_0) with the addresses $I^i := \mathsf{addr}(i)$. Its initial state s^i_0 contains a list of its domains and (private) SSL keys, a list of users and identites, and a private key for signing IDTokens. Besides this, the full state of i further contains a list of used nonces, and information about active sessions.

IdPs react to four types of requests:

First, they provide the `script_idp`, where a $t \in K_{\mathrm{id}}$ will be chosen and following requests to IdPs will be sent. IdP will transfer the data to RP by the communicating between two scripts `script_idp` and `script_rp` using `POSTMESSAGE`.

Second, they provide *IDToken* when receiving PID_{rp} and this PID_{rp} has already first. If not, IdPs will redirect to the login dialog.

After the user enter his username and password(secret) in the login dialog, a login request will send to `/authentication`. IdPs will check the parameters and set the login session.

The last type of requests IdPs react to is authorize requests with PID_{rp} and attribute scopes as parameters. After receving consent from browsers, IdPs will calculate PID_u and construct *IDToken*.

Formal description. In the following, we will first define the (initial) state of i formally and afterwards present the definition of the relation R^i.

To define the initial state, we will need a term that represents the "user database" of the IdP i. We will call this term $userset^i$. This database defines, which secret is valid for which identity. It is encoded as a mapping of identities to secrets. For example, if the secret $secret_1$ is valid for the identites $id_1$and the secret $secret_2$ is valid for the identity id_2, the $userset^i$ looks as follows:

$$userset^i = [id_1.\mathtt{username}{:}\langle id_1, secret_1\rangle, id_2.\mathtt{username}{:}\langle id_2, secret_2\rangle]$$

We define $userset^i$ as $userset^i = \langle\{\langle u.\mathtt{username}, \langle u, secret = \mathsf{secretOfID}(u)\rangle\rangle \mid u \in \mathsf{ID}^i\}\rangle$.

Definition 7. *A state $s \in Z^i$ of an IdP i is a term of the form $\langle sslkeys$, $users$, $signkey$, $sessions$, $corrupt\rangle$ where $sslkeys = sslkeys^i$, $users = userset^i$, $signkey \in \mathcal{N}$ (the key used by the IdP i to sign IDTokens), $sessions \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$.*

An initial state s_0^i of i is a state of the form $\langle sslkeys^i, userset^i, \mathsf{signkey}(i), \langle\rangle, \bot\rangle$.

The relation R^i that defines the behavior of the IdP i is defined as follows:

Algorithm 2 Relation of IdP R^r

```
Input: $\langle a, b, m\rangle, s$
1: let $s' := s$
2: if $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ then
3:     let $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
4:     let $m' := d_V(s')$
5:     let $a' := \mathsf{IPs}$
6:     stop $\langle a', a, m'\rangle, s'$
7: end if
8: let $m_{dec}, k, k', inDomain$ such that
     ↪ $\langle m_{\mathrm{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s'.\mathtt{sslkeys}$
     ↪ if possible; otherwise stop $\langle\rangle, s'$
```

```
9: let $n, method, path, parameters, headers, body$ such that
   ↪ $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m_{dec}$
   ↪ if possible; otherwise stop $\langle\rangle, s'$
10: if $path \equiv /script$ then
11:     let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{script\_idp}\rangle, k)$
12:     stop $\langle b, a, m'\rangle, s'$
13: else if $path \equiv /authentication$ then
14:     let $username := body[\texttt{username}]$
15:     let $password := body[\texttt{password}]$
16:     if $password \not\equiv s'.\texttt{userset}[username].secret$ then
17:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure}\rangle, k)$
18:         stop $\langle b, a, m'\rangle, s'$
19:     end if
20:     let $sessionid := \nu_2$
21:     let $s'.\texttt{sessions}[sessionid] := username$
22:     let $setCookie := \langle \texttt{Set-Cookie}, \langle\langle \texttt{sessionid}, sessionid, \top, \top, \top\rangle\rangle\rangle$
23:     let $m' := \langle \texttt{HTTPResp}, n, 200, \langle setCookie\rangle, \texttt{LoginSucess}\rangle$
24:     stop $\langle b, a, m'\rangle, s'$
25: else if $path \equiv /reqToken$ then
26:     let $cookie := headers[\texttt{Cookie}]$
27:     if $cookie[\texttt{sessionid}] \equiv \langle\rangle$ then
28:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unauthenticated}\rangle, k)$
29:         stop $\langle b, a, m'\rangle, s'$
30:     end if
31:     let $sessionid := cookie[\texttt{sessionid}]$
32:     let $PID_{rp} := parameters[\texttt{PID}_{\texttt{rp}}]$
33:     if $s'.\texttt{sessions}[sessionid].IDToken[PID_{rp}] \equiv \langle\rangle$ then
34:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unauthorized}\rangle, k)$
35:         stop $\langle b, a, m'\rangle, s'$
36:     end if
37:     let $IDToken := s'.\texttt{sessions}[sessionid].IDToken[PID_{rp}]$
38:     let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, IDToken\rangle, k)$
39:     stop $\langle b, a, m'\rangle, s'$
40: else if $path \equiv /authorize$ then
41:     let $cookie := headers[\texttt{Cookie}]$
42:     if $cookie[\texttt{sessionid}] \equiv \langle\rangle$ then
43:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unauthenticated}\rangle, k)$
44:         stop $\langle b, a, m'\rangle, s'$
45:     end if
46:     let $sessionid := cookie[\texttt{sessionid}]$
47:     let $PID_{RP} := parameters[\texttt{PID}_{\texttt{RP}}]$
48:     if $\texttt{IsValid}(PID_{RP}) \equiv \bot$ then
49:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle, k)$
50:         stop $\langle b, a, m'\rangle, s'$
51:     end if
52:     if $\texttt{IsInScope}(uid, body[\texttt{Attr}]) \equiv \bot$ then
53:         let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle, k)$
54:         stop $\langle b, a, m'\rangle, s'$
55:     end if
56:     let $u := s'.\texttt{sessions}[sessionid].u$
```

```
57:     let $ID_u := u.\mathtt{id}$
58:     let $PID_u := [ID_u]PID_{rp}$
59:     let $content := \langle PID_{rp}, PID_u \rangle$
60:     let $ver := \mathsf{sig}(content, s'.\mathtt{signkey})$
61:     let $IDToken := \langle content, ver \rangle$
62:     let $s'.\mathtt{sessions}[IDTokens] := s'.\mathtt{sessions}[IDTokens] +^{\langle\rangle} \langle PID_{rp}, IDToken \rangle$
63:     let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, IDToken \rangle, k)$
64:     stop $\langle b, a, m' \rangle, s'$
65: end if
66: stop $\langle\rangle, s'$
```

B.14 UPPRESSO Scripts

As already mentioned in Appendix B.1, the set $\mathcal{S}$ of the web system $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ consists of the scripts R^{att}, *script_rp*, *script_idp*, and with their string representations being `att_script`, `script_rp`, `script_idp`, and (defined by script).

In what follows, the scripts *script_rp* and *script_idp* are defined formally.

Relying Party Page (script_rp). As defined in SPRESSO, a script is a relation that takes a termas input and outputs a new term. The input term is provided by the browser. It contains the current internal state of the script (which we call *scriptstate* in what follows) and additional information containing all browser state information the script has access to, such as the input the script has obtained so far via XHRs and postMessages, information about windows, etc. The browser expects the output term to contain, among other information, the new internal *scriptstate*.

We first describe the structure of the internal scriptstate of the script *script_rp*.

Definition 8. *A* scriptstate s *of script_rp is a term of the form* $\langle phase, refXHR \rangle$*, where* $phase \in \mathbb{S}$*,* $refXHR \in \mathcal{N} \cup \{\bot\}$*.*

The initial scriptstate $initState_{rp}$ *of script_rp is* $\langle \mathtt{start}, \bot \rangle$*.*

We now specify the relation *script_rp* formally. We describe this relation by a non-deterministic algorithm.

Algorithm 3 Relation of *script_rp*

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage,$
    $\hookrightarrow$ $ids, secret \rangle$
1: let $s' := scriptstate$
2: let $command := \langle\rangle$
3: let $origin := \mathsf{GETORIGIN}(tree, docnonce)$
4: let $RPDomain := origin.\mathtt{host}$
5: switch $s'.\mathtt{phase}$ do
6:     case start:
7:         let $url := \langle \mathtt{URL}, \mathtt{S}, RPDomain, \mathtt{/loginSSO}, \langle\rangle \rangle$
8:         let $command := \langle \mathtt{HREF}, url, \mathtt{\_BLANK}, \langle\rangle \rangle$
9:         let $s'.\mathtt{phase} := \mathtt{expectt}$
```

```
    case $\mathtt{expectt}$:
        let $pattern := \langle \mathtt{POSTMESSAGE}, target, *, \langle \mathtt{t}, * \rangle \rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $t := \pi_2(\pi_4(input))$
            let $body := \langle \langle \mathtt{t}, t \rangle \rangle$
            let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{RPDomain}_{\mathtt{/startNegotiation}}, \mathtt{POST}, body,$
             $\hookrightarrow$ $s'.\mathtt{refXHR} \rangle$
            let $s'.\mathtt{phase} := \mathtt{expectCert}$
        end if
    case $\mathtt{expectCert}$:
        let $pattern := \langle \mathtt{XMLHTTPREQUEST}, *, s'.\mathtt{refXHR} \rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $Cert_{rp} := \pi_2(input).\mathtt{Cert}_{\mathtt{rp}}$
            let $IdPWindowNonce := \pi_1(\mathsf{SUBWINDOWS}(tree, docnonce)).\mathtt{nonce}$
            let $IdPOrigin := \mathsf{GETORIGIN}(tree, IdPWindowNonce)$
            let $command := \langle \mathtt{POSTMESSAGE}, IdPWindowNonce, \langle \mathtt{Cert}, Cert_{rp} \rangle,$
             $\hookrightarrow$ $IdPOrigin \rangle$
            let $s'.\mathtt{phase} := \mathtt{expectToken}$
        end if
    case $\mathtt{expectToken}$:
        let $pattern := \langle \mathtt{POSTMESSAGE}, target, *, \langle \mathtt{IDToken}, * \rangle \rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $IDToken := \pi_2(\pi_4(input))$
            let $body := \langle \langle \mathtt{IDToken}, IDToken \rangle \rangle$
            let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{RPDomain}_{\mathtt{/uploadToken}}, \mathtt{POST}, body,$
             $\hookrightarrow$ $s'.\mathtt{refXHR} \rangle$
            let $s'.\mathtt{phase} := \mathtt{expectLoginResult}$
        end if
    case $\mathtt{expectLoginResult}$:
        let $pattern := \langle \mathtt{XMLHTTPREQUEST}, *, s'.\mathtt{refXHR} \rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            if $\pi_2(input) \equiv \mathtt{LoginSuccess}$ then
                let Load Homepage
            end if
        end if
end switch
stop $\langle s', cookies, localStorage, sessionStorage, command \rangle$
```

Identity Provider Page (script_idp).

Definition 9. *A* scriptstate s of *script_idp is a term of the form* $\langle$*phase, user, parameters*$\rangle$ *with* $phase \in \mathbb{S}$*,* $user \in \mathsf{ID} \cup \{\langle\rangle\} \in \mathcal{T}$ *and* $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$*,. The* initial scriptstate *of script_idp is* $\langle \mathtt{start}, *, \langle\rangle \rangle$*.*

We now formally specify the relation of *script_idp*

Algorithm 4 Relation of *script_idp*

```
Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage,$
  $\hookrightarrow$ $ids, secret\rangle$
let $s' := scriptstate$
let $command := \langle\rangle$
let $target :=$ OPENERWINDOW$(tree, docnonce)$
let $origin :=$ GETORIGIN$(tree, docnonce)$
let $IdPDomain := origin.\texttt{host}$
switch $s'.\texttt{phase}$ do
    case $start$:
        let $t := \texttt{random}()$
        let $command := \langle \texttt{POSTMESSAGE}, target, \langle \texttt{t}, t\rangle, \langle\rangle\rangle$
        let $s'.\texttt{parameters}[t] := t$
        let $s'.\texttt{phase} := \texttt{expectCert}$
    case $\texttt{expectCert}$:
        let $pattern := \langle \texttt{POSTMESSAGE}, target, *, \langle \texttt{Cert}, *\rangle\rangle$
        let $input :=$ CHOOSEINPUT$(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $Cert_{rp} := \pi_2(\pi_4(input))$
            if $\mathsf{checksig}(Cert_{rp}.\texttt{ver}, Cert_{rp}.\texttt{content}, s'.\texttt{IdPConfig}.pubkey) \equiv \top$ then
                let $s'.\texttt{parameters}[cert] := Cert_{rp}$
                let $t := s'.\texttt{parameters}[t]$
                let $PID_{rp} := [t]Cert_{rp}.\texttt{content}[ID_{rp}]$
                let $s'.\texttt{parameters}[PID_{rp}] := PID_{rp}$
                let $body := \langle\langle \texttt{PID}_{\texttt{rp}}, PID_{rp}\rangle\rangle$
                let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\texttt{/reqToken}}, \texttt{POST}, body,$
                  $\hookrightarrow$ $s'.\texttt{refXHR}\rangle$
                let $s'.\texttt{phase} := \texttt{expectReqToken}$
            end if
        end if
    case $\texttt{expectReqToken}$:
        let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
        let $input :=$ CHOOSEINPUT$(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            if $\pi_2(input) \equiv \texttt{Unanthenticated}$ then
                let $s'.\texttt{user} \leftarrow ids$
                let $username := s'.\texttt{user}.name$
                let $password := \mathsf{secretOfID}(s'.\texttt{user})$
                let $body := \langle\langle \texttt{username}, username\rangle, \langle \texttt{password}, password\rangle\rangle$
                let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\texttt{/authentication}}, \texttt{POST}, body,$
                  $\hookrightarrow$ $s'.\texttt{refXHR}\rangle$
                let $s'.\texttt{phase} := \texttt{expectLoginResult}$
            else if $\pi_2(input) \equiv \texttt{Unauthorized}$ then
                let $PID_{rp} := s'.\texttt{parameters}[PID_{rp}]$
                let $Attr :=$ GETPARAMETERS$(tree, docnonce)[\texttt{iaKey}]$
                let $body := \langle\langle \texttt{PID}_{\texttt{rp}}, PID_{rp}\rangle, \langle \texttt{Attr}, Attr\rangle\rangle$
                let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\texttt{/authorize}}, \texttt{POST}, body,$
                  $\hookrightarrow$ $s'.\texttt{refXHR}\rangle$
                let $s'.\texttt{phase} := \texttt{expectToken}$
            else if  then
                let $IDToken := \pi_2(input)[\texttt{IDToken}]$
                let $RPOringin := \langle s'.\texttt{parameters}[cert].Content[\texttt{Enpt}], \texttt{S}\rangle$
                let $command := \langle \texttt{POSTMESSAGE}, target, \langle \texttt{IDToken}, IDToken\rangle, RPOrigin\rangle$
                let $s'.\texttt{phase} := \texttt{stop}$
            end if
        end if
    case expectLoginResult:
        let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            if $\pi_2(input) \equiv \texttt{LoginSuccess}$ then
                let $PID_{rp} := s'.\texttt{parameters}[PID_{rp}]$
                let $Attr := \mathsf{GETPARAMETERS}(tree, docnonce)[\texttt{iaKey}]$
                let $body := \langle\langle \texttt{PID}_{\texttt{rp}}, PID_{rp}\rangle, \langle \texttt{Attr}, Attr\rangle\rangle$
                let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\texttt{/authorize}}, \texttt{POST}, body,$
                 $\hookrightarrow$ $s'.\texttt{refXHR}\rangle$
                let $s'.\texttt{phase} := \texttt{expectToken}$
            end if
        end if
    case expectToken:
        let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $IDToken := \pi_2(input)[\texttt{IDToken}]$
            let $RPOringin := \langle s'.\texttt{parameters}[cert].Content[\texttt{Enpt}], \texttt{S}\rangle$
            let $command := \langle \texttt{POSTMESSAGE}, target, \langle \texttt{IDToken}, IDToken\rangle, RPOrigin\rangle$
            let $s'.\texttt{phase} := \texttt{stop}$
        end if
end switch
stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
```

C Proof of Security

We define the similar security properties as the definition 53 in SPRESSO. First note that the RP service token should be defined as $\langle IDToken,\ Acct\rangle$ which is $\langle n,\ i\rangle$ in SPRESSO. That is,

Definition 10. *let $\mathcal{UWS}^{auth}$ be an UPPRESSO web system for authentication analysis. We say that $\mathcal{UWS}^{auth}$ is secure if for every run ρ of $\mathcal{UWS}^{auth}$, every state $(S^j,\ E^j,\ N^j)$ in ρ, every $r \in \texttt{RP}$ that is honest, every RP service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r)$.*`serviceTokens`*, the following two conditions are satisfied:*

(A) If $\langle IDToken,\ Acct\rangle$ is derivable from the attackers knowledge in S^j (i.e., $\langle IDToken,\ Acct\rangle \in d_{\emptyset}(S^j(\texttt{attacker})))$, then it follows that the browser b owning Acct is fully corrupted in S^j (i.e., the value of isCorrupted is `FULLCORRUPT`*) or* `governor`*(Acct) is not an honest IdP (in S^j).*

(B) If the request corresponding to $\langle IDToken,\ Acct\rangle$ was sent by some

$b \in$ B which is honest in S^j, then b owns the ID_U which satisfies $Acct = [ID_U]S^j(r).ID_{RP}$.

To prove Theorem 5 in section 5.2, we are going to prove the following Lemmas. First we follows the Lemma 1, 2 and 3 in SPRESSO, which prove that the data transmitted through HTTPS is secure and the IdP's public key used for generating identity proof is secure. In UPPRESSO, only the single IdP is trusted, so that the public key is guaranteed to be always trusted. Therefore, we can also follow the proofs for Lemma 1, 2 and 3 in SPRESSO.

C.1 Proof of Property A

Then we prove the Property A is satisfied in UPPRESSO. As stated above, the Property A is defined as follows:

Definition 11. *Let $\mathcal{UWS}^{auth}$ be an UPPRESSO web system for authentication analysis. We say that $\mathcal{UWS}^{auth}$ is secure (with respect to Property A) if for every run rho of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in rho, every $r \in$ `RP` that is honest in S^j, every RP service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r)$.`serviceTokens` and derivable from the attackers knowledge in S^j (i.e., $\langle IDToken,\ Acct\rangle \in d_\emptyset(S^j($`attacker`$))$), it follows that the browser b owning Acct is fully corrupted in S^j (i.e., the value of isCorrupted is `FULLCORRUPT`) or `governor`(Acct) is not an honest IdP (in S^j).*

Same as the proof in SPRESSO, we want to show that every UPPRESSO web system is secure with regard to Property A and therefore assume that there exists an UPPRESSO web system that is not secure. We will lead this to a contradication and thereby show that all UPPRESSO web systems are secure (with regard to Property A).

In detail, we assume: *There is an UPPRESSO web system for authentication analysis $\mathcal{UWS}^{auth}$. We say that $\mathcal{UWS}^{auth}$ is secure (with respect to Property A) if for every run rho of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in rho, every $r \in$ `RP` that is honest in S^j, every RP service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r)$.`serviceTokens` and derivable from the attackers knowledge in S^j (i.e., $\langle IDToken,\ Acct\rangle \in d_\emptyset(S^j($`attacker`$))$), it follows that the browser b owning Acct is not fully corrupted in S^j and `governor`(Acct) is an honest IdP (in S^j).*

We now proceed to to proof that this is a contradiction. Let $I :=$ `governor`(i). We know that I is an honest IdP. As such, it never leaks its signing key (see Algorithm 2). Therefore, the signed subterm $Content := \langle PID_{RP}, PID_U, s'.Issuer, Validity\rangle$, $Sig := SigSign(Content, s'.SK)$ and $IDToken := \langle Content, Sig\rangle$ had to be created by the IdP I. An (honest) IdP creates signatures only in Line 48-50 of Algorithm 2.

Lemma 1. *(Same as Lemma 4 in SPRESSO) Under the assumption above, only the browser b can issue a request req that triggers the IdP I to create the signed term IDToken. The request was sent by b over HTTPS using I's public HTTPS key.*

Proof. The proof is same as the Lemma 4's proof in SPRESSO. It can be proved that the $IDToken$ only contains the $PID_U := [ID_U]PID_{RP}$ while PID_U is provided by b, and b owns the password of ID_U. □

Lemma 2. *(Same as Lemma 5 in SPRESSO) In the browser b, the request req was triggered by script_idp loaded from the origin $\langle d, S\rangle$ for some $d \in \mathtt{dom}(I)$.*

Proof. The proof follows the Lemma 5's proof in SPRESSO. It can be proved that only the IdP's script *script_idp* owns the password of ID_U can request the $IDToken$ from I. □

Lemma 3. *(Same as Lemma 6 in SPRESSO) In the browser b, the script script_idp receives the response to the request req (and no other script), and at this point, the browser is still honest.*

Proof. The proof follows Lemma 6's proof in SPRESSO. It is proved that only the closed-corrupted browser cannot receive the $IDToken$ responding to the req started by the honest browser b. □

Lemma 7 in SPRESSO is not useful here because there is no FWD server in UPPRESSO.

Lemma 4. *(Same as Lemma 8 in SPRESSO) The script script_idp forwards the IDToken only to the script script_rp loaded from the origin $\langle d_r, S\rangle$.*

Proof. The proof is same as proof of Lemma 8 in SPRESSO. It can be proved that, the $IDToken$ held by the honest *script_idp* is only sent to the origin $\langle Cert_{RP}.Enpt_{RP}, S\rangle$, while the $IDToken.PID_{RP} \equiv [t]Cert_{RP}.ID_{RP}$, and t is the one-time random number. The relation of ID_RP and $Enpt$ is guaranteed by the signature generated by IdP I. The process is shown at Line 9, 16, 19, 21, 38, 39, 59, 60 in Algorithm **??**. □

Lemma 5. *(Same as Lemma 9 in SPRESSO) From the RP document, the IDToken is only sent to the RP r and over HTTPS*

Proof. The proof follows the proof of Lemma 9 in SPRESSO. It is proved that *script_rp* of the origin $\langle Cert_{RP}.Enpt_{RP}, S\rangle$ would only sent to the corresponding RP r, which is shown in Algorithm **??**. □

The proofs show that the $IDToken$ is only sent to the honest browser (Lemma 1-7) and target RP (Lemma 8-9). Above proofs can be reduced to the Confidentiality and Integrity Properties, simply described as the Theorem 3 and 4 in section 5.2. These proofs are enough for SPRESSO system to show its security, however, they are not enough for UPPRESSO. So far, the proofs only guarantee that the $IDToken$ must be sent to the target RP. In SPRESSO, as the tag can be only decrypted to unique $Domain$, the target RP must be the honest RP (the target of an adversary). However, in UPPRESSO, while an RP receives an $IDToken$, he may try to use this token to login another honest RP, as long as he can find the $t^{adversary}$ satisfied $IDToken.PID_{RP} \equiv [t^{adversary}]ID_{RP}^{honest}$. Therefore, the following Lemma should be proved.

Lemma 6. *The $t^{adversary}$ is not derivable from the attackers knowledge in S^j (i.e., $\langle IDToken,\ Acct\rangle \in d_{\emptyset}(S^j(\texttt{attacker}))$), which satisfies that $IDToken.PID_{RP} \equiv [t^{adversary}]ID_{RP}^{honest}$.*

Proof. This Lemma can be proved by the Theorem 1 in section 5.2, as the RP Designation Property. □

Therefore, there is a contradication to the assumption, where we assumed that $\langle IDToken,\ Acct\rangle \in d_{\emptyset}(S^j(\texttt{attacker}))$. This shows every $\mathcal{UWS}^{auth}$ is secure in the sense of Property A.

C.2 Proof of Property B

As stated above, Property B is defined as follows:

Definition 12. *Let $\mathcal{UWS}^{auth}$ be an UPPRESSO web system for authentication analysis. We say that $\mathcal{UWS}^{auth}$ is secure (with respect to Property A) if for every run rho of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in rho, every $r \in \texttt{RP}$ that is honest in S^j, every RP service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r).\texttt{serviceTokens}$, with the request corresponding to $\langle IDToken,\ Acct\rangle$ sent by some $b \in B$ which is honest in S^j, b owns Acct.*

First we follows the Lemma 10 and its proof in SPRESSO, which guarantees that the request corresponding to $\langle IDToken,\ Acct\rangle$ sent by honest b is loaded from *script_rp*. Then we are going to prove the $IDToken$ uploaded by honest b can only be related with the *Acct* owned by b (which is quite different from SPRESSO).

Lemma 7. *For every $IDToken$ uploaded by honest b during authentication, the honest $r \in RP$ can always derive the service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r).\texttt{serviceTokens}$, where b owns Acct.*

Proof. The RP accepts the user's identity at Line 43 in Algorithm 1. And the identity is generated at Line 38, based on the PID_U retrieved from the $IDTpken$ and the trapdoor t^{-1}. The t^{-1} is generated at Line 13, set at Line 14, and never changed, as the multiplicative inverse of t. The $IDToken$ is issued at Line 50 in Algorithm 2. The IdP generates the PID_U based on the PID_{RP} and ID_U related to b *in*$\texttt{Browser}$.

An attacker may allure the honest user to upload the $IDToken \in d_{\emptyset}(S^j(\texttt{attacker}))$ to honest $r \in \texttt{RP}$, so that there may be $Acct \in d_{\emptyset}(S^j(\texttt{attacker}))$. However, while b has already negotiated the PID_{RP} with r, the opener of the *script_idp* must be the *script_rp*. As the t generated at Line 7, Algorithm **??**, and PID_{RP} generated at Line 21 in Algorithm **??**. The t is only sent to *script_rp* at Line 8 in Algorithm **??**, and the *script_rp* receives it at Line 18 in Algorithm **??**. The PID_{RP} is sent to the honest IdP at Lines 23 and 50 in Algorithm **??**, which is used for generating the $IDToken$.

For every $IDToken$ sent by honest b and honest r, there must be $IDToken.PID_{RP} \equiv [t]Cert_{RP}.ID_{RP}$, $IDToken.PID_U \equiv$

$[ID_U]IDToken.PID_U$ and $Acct \equiv [t^{-1}]IDToken.PID_U$. According to the proof of Theorem 2 in section 5.2, the $Acct$ must be owned by honest b ($Acct \equiv [ID_U]S^j(r).ID_{RP}$, where ID_U is related to b), which can be define as the User Identification Property . $\square$

With the above proofs, we now can guarantee that every $\mathcal{UWS}^{auth}$ system satisfies the requirements in Definition 12, therefore $\mathcal{UWS}$ must be secure of Property B.

D Proof of Privacy

In our privacy analysis, we show that an identity provider in UPPRESSO cannot learn where its users log in. We formalize this property as an indistinguishability property: an identity provider (modeled as a web attacker) cannot distinguish between a user logging in at one relying party and the same user logging in at a different relying party.

D.1 Formal Model of UPPRESSO for Privacy Analysis

Definition 13 (Challenge Browser)**.**

Definition 14 (Deterministic DY Process)**.**

Definition 15 (UPPRESSO Web System for Privacy Analysis)**.** *Let* $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, script, E^0)$ *be an UPPRESSO web system with* $\mathcal{W} = Hon \cup Web \cup Net$, $Hon = B \cup RP \cup IDP \cup DNS$. *Let* $attacker \in Web$ *be some web attacker. Let* dr *be a domain of* r_1 *or* r_2 *and* $b(dr)$ *be a challenge browser. Let* $Hon\prime := \{b(dr)\} \cup RP \cup DNS$, $Web\prime := Web$, *and* $Net\prime := \emptyset$. *Let* $\mathcal{W}\prime := Hon\prime \cup Web\prime \cup Net\prime$. *We call* $\mathcal{UWS}^{priv}(dr) = (\mathcal{W}\prime, \mathcal{S}\prime, script\prime, E^0, attacker)$ *an UPPRESSO web system for privacy analysis.*

Definition 16 (IdP-Privacy)**.** *Let*

$$\begin{aligned} \mathcal{UWS}^{priv}_1 &:= \mathcal{UWS}^{priv}(dr_1) = (\mathcal{W}_1, \mathcal{S}, script, E^0, attacker_1) \\ \mathcal{UWS}^{priv}_2 &:= \mathcal{UWS}^{priv}(dr_2) = (\mathcal{W}_2, \mathcal{S}, script, E^0, attacker_2) \end{aligned} \tag{1}$$

be UPPRESSO web systems for privacy analysis. We say that $\mathcal{UWS}^{priv}$ *is IdP-private iff* $\mathcal{UWS}^{priv}_1$ *and* $\mathcal{UWS}^{priv}_2$ *are indistinguishable.*

D.2 Definition of Equivalent Configurations

Let $\mathcal{UWS}^{priv}_1$ and $\mathcal{UWS}^{priv}_2$ be UPPRESSO web system for privacy analysis. Let (S_1, E_1, N_1) be a configuration of $\mathcal{UWS}^{priv}_1$ and (S_2, E_2, N_2) accordingly.

Definition 17 (Challenge Browser)**.**

Definition 18 (Term Equivalence up to Proto-Tags)**.**

Definition 19 (Equivalence of HTTP Requests)**.**

Definition 20 (Extracting Entries from Login Sessions)**.**

Definition 21 (Login Session Token)**.**

Definition 22 (Equivalence of States)**.** *Same as Definition 79 in SPRESSO except that the first condition in Definition 79 in SPRESSO is not applicable.*

Definition 23 (Equivalence of Events)**.** *Same as Definition 80 in SPRESSO except that the forth condition in Definition 80 in SPRESSO is not applicable.*

Definition 24 (Equivalence of Configurations)**.**

D.3 Privacy Proof

Theorem 1. *Every UPPRESSO web system for privacy analysis is IdP-private.*

Let $\mathcal{UWS}^{priv}$ be UPPRESSO web system for privacy analysis.

To prove Theorem 1, we have to show that the UPPRESSO web systems $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$ are indistinguishable. To show the indistinguishability of $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$, we show that they are indistinguishable under all schedules σ. For this , we first note that for all σ, there is only one run induced by each σ(as our web system, when scheduled, is deterministic). We now proceed to show that for all schedules $\sigma = (\zeta_1, \zeta_2, \dots)$, iff σ induces a run $\sigma(\mathcal{UWS}_1^{priv})$ there exists a run $\sigma(\mathcal{UWS}_2^{priv})$ such that $\sigma(\mathcal{UWS}_1^{priv}) \approx \sigma(\mathcal{UWS}_1^{priv})$

We now show that if two configurations are α-equivalent, then the view of the attacker is statically equivalent.

Lemma 8. *(Same as Lemma 12 in SPRESSO) Let (S_1, E_1, N_1) and (S_2, E_2, N_2) be two α-equivalent configurations. Then $S_1(attacker) \approx S_2(attacker)$.*

Lemma 9. *(Same as Lemma 13 in SPRESSO) The initial configurations (S_1^0, E^0, N^0) of $\mathcal{UWS}_1^{priv}$ and (S_2^0, E^0, N^0) of $\mathcal{UWS}_2^{priv}$ are α-equivalent.*

Proof. Let $\theta = H = L = \emptyset$.Obviously, both latter conditions are true. For all parties $p \in \mathcal{W}_1 \setminus \{b_1\}$, it is clear that $S_1^0(p) = S_2^0(p)$. Also the states $S_1^0(b_1) = S_2^0(b_2)$ are equal. Therefore, all conditions of Definition 22 are fulfilled. Hence, the initial configurations are α-equivalent. □

Lemma 10. *(Same as Lemma 14 in SPRESSO) Let (S_1, E_1, N_1) and (S_2, E_2, N_2) be two α-equivalent configurations of $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$, respectively. Let $\zeta = \langle ci, cp, \tau_{process}, cmd_{switch}, cmd_{window}, \tau_{script}, url\rangle$ be a web system command. Then, ζ induces a processing step in either both configurations or in none. In the former case, let $(S_1\prime, E_1\prime, N_1\prime)$ and $(S_2\prime, E_2\prime, N_2\prime)$ be configurations induced by ζ such that*

$$(S_1, E_1, N_1) \xrightarrow{\zeta} (S_1\prime, E_1\prime, N_1\prime) and (S_2, E_2, N_2) \xrightarrow{\zeta} (S_2\prime, E_2\prime, N_2\prime) \quad (2)$$

Then $(S_1\prime, E_1\prime, N_1\prime)$ and $(S_2\prime, E_2\prime, N_2\prime)$ are α-equivalent.

Proof. Let θ be a set of proto-tags and H be a set of nonces for which α-equivalence holds and let $L := \bigcup_{a\in\theta} \text{loginSessionTokens}(a, S_1, S_2)$,$K := \{k|\exists n : enc_s(\langle y, n\rangle, k) \in \theta\}$

To induce a processing step, the ci-th message from E_1 or E_2, respectively, is selected.Following Definition 23, we denote these messages by $e_i^{(1)}$ or $e_i^{(2)}$, respectively. We now differentiate between the receivers of the messages by denoting the induced processing steps by

$$\begin{aligned} (S_1, E_1, N_1) &\xrightarrow[p_1 \to E_{out}^{(1)}]{\langle a_1, f_1, m_1\rangle \to p_1} (S_1\prime, E_1\prime, N_1\prime) \\ (S_2, E_2, N_2) &\xrightarrow[p_2 \to E_{out}^{(2)}]{\langle a_2, f_2, m_2\rangle \to p_2} (S_2\prime, E_2\prime, N_2\prime) \end{aligned} \tag{3}$$

Case $p_1 = dns$: In this case, only Cases 1a, 1b and 1c of Definition 80 can apply. Hence, $p_2 = dns$.

(*):As both events are static except for IP addresses, the HTTP nonce, and the HTTPS key, there is no k contained in the input messages(except potentially in tags, from where it cannot be extracted), and the output messages are sent to f_1 or f_2, respectively, they can not cantian any $l \in L$ or $k \in K$. Hence, Condition 2 of Definition 80 holds true.

We note that (*) so-called Condition 2 applies analogously in cases 1a, 1b and 1c. In the case 1a, it is easy to see that $E_{out}^{(1)} \rightleftharpoons_\theta E_{out}^{(2)}$.In the case 1c, it is easy to that the DNS server only outputs empty events in both processing steps. In the case 1b, $E_{out}^{(1)}$ and $E_{out}^{(2)}$ are such that Case 1d of Definition 80 applies.

Therefore, $E_1\prime$ and $E_2\prime$ are β-equivalent under (θ, H, L) in all three cases. As there are no changes to any state in all cases, we have that $S_1\prime$ and $S_2\prime$ are γ-equivalent under (θ, H). No new nonces are chosen, hence $N_1\prime = N_1 = N_2 = N_2\prime$.

Case $p_1 = r_1$: In this case, we only distinct several cases of HTTP(S) requests that can happen. The others are ignored the same as SPRESSO.

There are four possible types of HTTP requests that are accepted by r_1 in Algorithm 1:

- path=/script(get the rp-script), Line 3;
- path=/loginSSO(start a login), Line 6;
- path=/startNegotiation(derive a PID_rp), Line 9;
- path=/uploadToken(verify ID token, calculate Acct), Line 18.

From the cases in Definition 23, only two can possibly apply here:Case 1a and Case 1e. For both cases, we will now analyze each of the HTTP requests listed above separately.

Definition 23,Case 1a:$e_i^{(1)} \rightleftharpoons e_i^{(2)}$. This case implies $p_2 = r_1 = p_1$. As we see below, for the output events $E_{out}^{(1)}$ and $E_{out}^{(2)}$ (if any) only Case 1a of Definition 23 applies. This implies the nonce of both the incoming HTTP requests and HTTP responses cannot be in H.

- path=/script In this case, the same output event is produced whose message is
$$\langle HTTPResp, n, 200, \langle\rangle, RPScript\rangle \tag{4}$$
We can note that Condition 5 of Definition 23 holds true and, also, (*) applies.The remaining conditions are trivially fulfilled and $E_1\prime$ and $E_2\prime$ are β-equivalent under (θ, H, L).As there are no changes to any state, we have that $S_1\prime$ and $S_2\prime$ are γ-equivalent under (θ, H). No new nonces are chosen, hence $N_1\prime = N_1 = N_2 = N_2\prime$.

- path=/loginSSO In this case, the reason for equivalence holding is similar to the case above since the same output event is produced.

- path=/startNegotiation(derive a PID_rp), Line 9;

- path=/uploadToken(verify ID token, calculate Acct), Line 18.

$\square$