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, however, we have some difference from SPRESSO. In particular, we remove some processes and add some function symbols for asymmetric encryption/decryption.

A.1 Functions Symbols

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.
- $\mathtt{isValid}(P)$ 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.

A.2 DNS servers

In SPRESSO, when receiving an e-mail address, RP needs to send DNS requests to DNS servers manually to fetch the information of the IdP server. Since there may be various DNS servers in SPRESSO, DNS server security issues need to be given special consideration. As a result, DNS servers are added into the formal model of SPRESSO.

In UPPRESSO, however, we only have one centralized IdP server, and all RPs know the relevant information of the IdP in advance. So all DNS requests are generated spontaneously by the browser, not introduced by our scripts. Therefore, we remove DNS servers from the formal model of UPPRESSO.

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 containing only one identity provider with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. 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.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
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.

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

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, the only one identity provider in IDP, 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, the only one identity provider in IDP, every web attacker in Web, and every network attacker in Net. Browsers (in B) 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}, an infinite set K_{point}, and a finite set $\mathsf{Secrets}$. We thus have

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

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 $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{SSL}}$ 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_{point} contains all valid points on the curve. The set K_{point} 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

There are many different types of identities in UPPRESSO. The first is browsers' identities at the IdP. Browsers share a $\mathit{username} \in \mathbb{S}$ with IdP to identify an user uniquely, and not like SPRESSO, UPPRESSO doesn't need a domain to identify the IdP.

By $\mathsf{secretOfName} : \mathbb{S} \to \mathsf{Secrets}$ we denote the bijective mapping that assigns secrets to all usernames.

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 $\mathsf{ownerOfUser} : \mathbb{S} \to \mathsf{B}$, $\mathit{username} \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfName}(\mathit{username}))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).

Besides, the browsers' identities also have IDs $\mathit{ID}_u := u \in [1, n)$ which is only known to the IdP. By $\mathsf{IDOfName} : \mathbb{S} \to N$ we denote the bijective mapping that assigns IDs to all usernames.

The second type of identities is relying parties' identities at the IdP, which are IDs $\mathit{ID}_{rp} := [r]G \in K_{\text{point}}$ in which $r \in [1, n)$ is unknown to any party.

The third type of identities is browsers' identities at the Relying Parties which is called $\mathit{Acct} := [\mathit{ID}_u]\mathit{ID}_{rp} = [ur]G \in K_{\text{point}}$.

B.5 Tags, Identity Tokens and Service Tokens

Definition 1. *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 2. *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 $k \in K_{sign}$.*

Definition 3. *A* service token *is a term of the form $\langle nonce, Acct\rangle$ with $Acct = [t^{-1}]PID_u = [t^{-1}][utr]G = [ur]G$ for a nonce t.*

B.6 Corruption

RPs 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 is *honest* if the according part of their state ($s.\mathtt{corrupt}$) is $\perp$, and that they are corrupted otherwise.

IdP is always honest-but-curious and never corrupted.

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 $\mathcal{W}$ 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.

B.8 TLS 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 TLS keys: For an atomic DY process p we define

$$\mathit{tlskeys}^p = \langle \{ \langle d, \mathsf{tlskey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle.$$

B.9 Web Attackers

Each $wa \in \mathsf{Web}$ is a web attacker. The initial state of each wa is $s_0^{wa} = \langle \mathit{attdoms}, \mathit{tlskeys}, \mathit{signkeys} \rangle$, where $\mathit{attdoms}$ is a sequence of all domains along with the corresponding private keys owned by wa, $\mathit{tlskeys}$ is a sequence of all domains and the corresponding public keys, and *signkeys* contains the public signing key for the IdP.

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 $s_0^{na} = \langle \mathit{attdoms}, \mathit{tlskeys}, \mathit{signkeys} \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker na, *tlskeys* is a sequence of all domains and the corresponding public keys, and *signkeys* contains the public signing key for the IdP.

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 $U^b := \mathsf{ownerOfUser}^{-1}(b)$ be the set of all usernames of b, Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (tls) key, according to the mapping tlskey; the DNS address is $\mathsf{addr}(p)$ with $p \in \mathcal{W}$; *ids* is $\langle U^b \rangle$; *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. 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, 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 $\mathit{IDToken}.\texttt{PID}_{\texttt{rp}}$ with the PID_{rp} in the login session, and checks whether $\mathit{IDToken}.\texttt{PID}_{\texttt{ver}}$ 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 4. *A* login session record *is a term of the form* $\langle t, PID_{rp}\rangle$ *with* $t \in N, PID_{rp} \in K_{point}$.

Definition 5. *A* state $s \in Z^r$ of an RP *is a term of the form* $\langle \mathit{keyMapping}, \mathit{tlskeys}, \mathit{loginSessions}, \mathit{serviceTokens}, \mathit{corrupt}, \mathit{IdPConfig}, \mathit{rp}\rangle$ *where* $\mathit{keyMapping} \in [\mathbb{S} \times \mathcal{N}]$, $\mathit{tlskeys} = \mathit{tlskeys}^r$, $\mathit{serviceTokens} \in [\mathcal{N} \times K_{point}]$, $\mathit{loginSessions} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ *is a dictionary of login session records,* $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{IdPConfig} \in \mathcal{T}_{\mathcal{N}}$ *is the configuration retrieved from IdP server,* $\mathit{rp} \in K_{point}$ *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* $s_0^r.\texttt{serviceTokens} = s_0^r.\texttt{loginSessions} = \langle\rangle$, $s_0^r.\texttt{corrupt} = \perp$, $s_0^r.\texttt{keyMapping}$ *is the same as the keymapping for browsers above,* $s_0^r.\texttt{IdPConfig} = \langle \mathit{pubkey}, \mathit{scriptUrl}, \mathit{Cert}_{rp}\rangle$ *and* $s_0^r.\texttt{rp} = [r]G$ *with* $r \in N$.

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 $s'.\texttt{corrupt} \not\equiv \perp \vee m \equiv \texttt{CORRUPT}$ then
 3:     let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
 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
      $\hookrightarrow$ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s'.\texttt{sslkeys}$
      $\hookrightarrow$ if possible; otherwise stop $\langle\rangle, s'$
 9: let $n, method, path, parameters, headers, body$ such that
      $\hookrightarrow$ $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m_{dec}$
      $\hookrightarrow$ if possible; otherwise stop $\langle\rangle, s'$
10: if $path \equiv /script$ then
11:     let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{script\_rp}\rangle, k)$
12:     stop $\langle b, a, m'\rangle, s'$
13: else if $path \equiv /loginSSO$ then
14:     let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 302, \langle\langle \texttt{Location}, s'.\texttt{IdPConfig}.scriptUrl\rangle\rangle, \langle\rangle\rangle, k)$
15:     stop $\langle b, a, m'\rangle, s'$
16: else if $path \equiv /startNegotiation$ then
```

```
        let $\mathit{loginSessionToken} := \nu_1$
        let $t := \mathit{body}[t]$
        let $\mathit{ID}_{rp} := s'.\mathtt{rp}$
        let $\mathit{PID}_{rp} := [t]\mathit{ID}_{rp}$
        let $\mathit{state} := \mathtt{expectToken}$
        let $\mathit{Cert}_{rp} := s'.\mathtt{IdPConfig}.\mathit{Cert}_{RP}$
        let $s'.\mathtt{loginSessions}[\mathit{loginSessionToken}] := \langle t, \mathit{PID}_{rp}, \mathit{state}\rangle$
        let $\mathit{setCookie} := \langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{sessionid}, \mathit{loginSessionToken}, \top, \top, \top\rangle\rangle\rangle$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle \mathit{setCookie}\rangle, \langle \mathtt{Cert_{RP}}, \mathit{Cert}_{rp}\rangle\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    else if $\mathit{path} \equiv /\mathit{uploadToken}$ then
        let $\mathit{cookie} := \mathit{headers}[\mathtt{Cookie}]$
        if $\mathit{cookie}[\mathtt{sessionid}] \equiv \langle\rangle$ then
            stop $\langle\rangle, s'$
        end if
        let $\mathit{loginSessions} := s'.\mathtt{loginSessions}[\mathit{cookie}[\mathtt{sessionid}]]$
        if $\mathit{loginSessions}.\mathtt{state} \not\equiv \mathit{expectToken}$ 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 $s'.\mathtt{loginSessions} := s'.\mathtt{loginSessions} - \mathit{body}[\mathit{loginSessionToken}]$
        let $\mathit{IDToken} := \mathit{body}[\mathtt{IDToken}]$
        if $\mathit{IDToken}.\mathtt{PID_{rp}} \not\equiv \mathit{loginSessions}.\mathtt{PID_{rp}}$ 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 $\mathsf{checksig}(\mathit{IDToken}.\mathtt{ver}, \langle \mathit{IDToken}.\mathtt{PID_{rp}}, \mathit{IDToken}.\mathtt{PID_u}\rangle, s'.\mathtt{IdPConfig}.\mathit{pubkey}) \equiv \bot$
        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 $\mathit{PID}_u := \mathit{IDToken}.\mathtt{PID_u}$
        let $\mathit{Acct} := [\mathit{loginSessions}.\mathtt{t}]\mathit{PID}_u$
        let $\mathit{nonce} := \nu_2$
        let $s'.\mathtt{serviceTokens} := s'.\mathtt{serviceTokens} +^{\langle\rangle} \langle \mathit{nonce}, \mathit{Acct}\rangle$
        let $\mathit{setCookie} := \langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{sessionid}, \mathit{nonce}, \top, \top, \top\rangle\rangle\rangle$
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle \mathit{setCookie}\rangle, \mathtt{LoginSuccess}\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    end if
    stop $\langle\rangle, s'$
```

B.13 Identity Providers

The identity provider IdP is a web server modeled as an atomic process (I, Z, R, s_0) with the addresses $I := \mathsf{addr}(\mathsf{IdP})$. Its initial state s_0 contains a list of its domains and (private) TLS keys, a list of users and identites, and a private key for signing IDTokens. Besides this, the full state of IdP further contains a list of used nonces, and information about active sessions.

IdP react to four types of requests:

First, they provide the `script_idp`, where a t will be chosen and following requests to IdP 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 IdP formally and afterwards present the definition of the relation R.

To define the initial state, we will need a term that represents the "user database" of the IdP. We will call this term *userset*. This database defines, which secret and ID_u is valid for which identity. It is encoded as a mapping of username to secrets and ID_u. For example, if the secret $secret_1$ and ID_{u_1} is valid for the username u_1 and the secret $secret_2$ and ID_{u_2} is valid for the identity u_2, the $userset^i$ looks as follows:

$$userset = [u_1{:}\langle ID_{u_1}, secret_1\rangle, u_2{:}\langle ID_{u_2}, secret_2\rangle]$$

We define *userset* as $userset = \langle\{\langle username, \langle id = \mathsf{IDOfName}(username), secret = \mathsf{secretOfName}(username)\rangle$ $\mathbb{S}\}\rangle$.

Definition 6. *A* state $s \in Z$ of the IdP *is a term of the form* $\langle tlskeys, users, signkey, sessions, corrupt\rangle$ *where* $tlskeys = tlskeys$, $users = userset$, $signkey \in \mathcal{N}$ *(the key used by the IdP 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 *of* IdP *is a state of the form* $\langle tlskeys, userset, \mathsf{signkey}(\mathsf{IdP}), \langle\rangle, \bot\rangle$.

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

Algorithm 2 Relation of IdP 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
      ↪ $\langle \mathtt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m_{dec}$
      ↪ if possible; otherwise stop $\langle\rangle, s'$
```

```
if $path \equiv /script$ then
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{script\_idp}\rangle, k)$
    stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /authentication$ then
    let $username := body[\mathtt{username}]$
    let $password := body[\mathtt{password}]$
    if $password \not\equiv s'.\mathtt{userset}[username].secret$ then
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{LoginFailure}\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    end if
    let $sessionid := \nu_3$
    let $s'.\mathtt{sessions}[sessionid] := username$
    let $setCookie := \langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{sessionid}, sessionid, \top, \top, \top\rangle\rangle\rangle$
    let $m' := \langle \mathtt{HTTPResp}, n, 200, \langle setCookie\rangle, \mathtt{LoginSucess}\rangle$
    stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /reqToken$ then
    let $cookie := headers[\mathtt{Cookie}]$
    if $cookie[\mathtt{sessionid}] \equiv \langle\rangle$ then
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Unauthenticated}\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    end if
    let $sessionid := cookie[\mathtt{sessionid}]$
    let $PID_{rp} := parameters[\mathtt{PID_{rp}}]$
    if $s'.\mathtt{sessions}[sessionid].IDToken[PID_{rp}] \equiv \langle\rangle$ then
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Unauthorized}\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    end if
    let $IDToken := s'.\mathtt{sessions}[sessionid].IDToken[PID_{rp}]$
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, IDToken\rangle, k)$
    stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /authorize$ then
    let $cookie := headers[\mathtt{Cookie}]$
    if $cookie[\mathtt{sessionid}] \equiv \langle\rangle$ then
        let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, \mathtt{Unauthenticated}\rangle, k)$
        stop $\langle b, a, m'\rangle, s'$
    end if
    let $sessionid := cookie[\mathtt{sessionid}]$
    let $PID_{RP} := parameters[\mathtt{PID_{RP}}]$
    if $\mathtt{IsValid}(PID_{RP}) \equiv \bot$ 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
    if $\mathtt{IsInScope}(uid, body[\mathtt{Attr}]) \equiv \bot$ 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 $username := s'.\mathtt{sessions}[sessionid].username$
    let $ID_u := s'.\mathtt{userset}[username].id$
    let $PID_u := [ID_u]PID_{rp}$
    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 $s'.\mathtt{sessions}[IDToken] := s'.\mathtt{sessions}[IDTokens] +^{\langle\rangle} \langle PID_{rp}, IDToken \rangle$
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 7. *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 $s'.\mathtt{phase} := \mathtt{expectt}$
10:    case expectt:
11:        let $pattern := \langle \mathtt{POSTMESSAGE}, target, *, \langle \mathtt{t}, * \rangle \rangle$
12:        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
```

```
13:     if $input \not\equiv \bot$ then
14:         let $t := \pi_2(\pi_4(input))$
15:         let $body := \langle\langle \mathtt{t}, t\rangle\rangle$
16:         let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{RPDomain}_{\mathtt{/startNegotiation}}, \mathtt{POST}, body,$
             $\hookrightarrow\ s'.\mathtt{refXHR}\rangle$
17:         let $s'.\mathtt{phase} := \mathtt{expectCert}$
18:     end if
19:   case expectCert:
20:     let $pattern := \langle \mathtt{XMLHTTPREQUEST}, *, s'.\mathtt{refXHR}\rangle$
21:     let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
22:     if $input \not\equiv \bot$ then
23:         let $Cert_{rp} := \pi_2(input).\mathtt{Cert_{rp}}$
24:         let $IdPWindowNonce := \pi_1(\mathsf{SUBWINDOWS}(tree, docnonce)).\mathtt{nonce}$
25:         let $IdPOrigin := \mathsf{GETORIGIN}(tree, IdPWindowNonce)$
26:         let $command := \langle \mathtt{POSTMESSAGE}, IdPWindowNonce, \langle \mathtt{Cert}, Cert_{rp}\rangle,$
             $\hookrightarrow\ IdPOrigin\rangle$
27:         let $s'.\mathtt{phase} := \mathtt{expectToken}$
28:     end if
29:   case expectToken:
30:     let $pattern := \langle \mathtt{POSTMESSAGE}, target, *, \langle \mathtt{IDToken}, *\rangle\rangle$
31:     let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
32:     if $input \not\equiv \bot$ then
33:         let $IDToken := \pi_2(\pi_4(input))$
34:         let $body := \langle\langle \mathtt{IDToken}, IDToken\rangle\rangle$
35:         let $command := \langle \mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{RPDomain}_{\mathtt{/uploadToken}}, \mathtt{POST}, body,$
             $\hookrightarrow\ s'.\mathtt{refXHR}\rangle$
36:         let $s'.\mathtt{phase} := \mathtt{expectLoginResult}$
37:     end if
38:   case expectLoginResult:
39:     let $pattern := \langle \mathtt{XMLHTTPREQUEST}, *, s'.\mathtt{refXHR}\rangle$
40:     let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
41:     if $input \not\equiv \bot$ then
42:         if $\pi_2(input) \equiv \mathtt{LoginSuccess}$ then
43:             let Load Homepage
44:         end if
45:     end if
46: end switch
47: stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
```

Identity Provider Page (script_idp).

Definition 8. *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* $parameters \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$,. *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 $target := \mathsf{OPENERWINDOW}(tree, docnonce)$
let $origin := \mathsf{GETORIGIN}(tree, docnonce)$
let $IdPDomain := origin.\mathtt{host}$
switch $s'.\mathtt{phase}$ do
    case $start$:
        let $t := \mathtt{random}()$
        let $command := \langle\mathtt{POSTMESSAGE}, target, \langle\mathtt{t}, t\rangle, \langle\rangle\rangle$
        let $s'.\mathtt{parameters}[t] := t$
        let $s'.\mathtt{phase} := \mathtt{expectCert}$
    case $\mathtt{expectCert}$:
        let $pattern := \langle\mathtt{POSTMESSAGE}, target, *, \langle\mathtt{Cert}, *\rangle\rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            let $Cert_{rp} := \pi_2(\pi_4(input))$
            if $\mathsf{checksig}(Cert_{rp}.\mathtt{ver}, Cert_{rp}.\mathtt{content}, s'.\mathtt{IdPConfig}.pubkey) \equiv \top$ then
                let $s'.\mathtt{parameters}[cert] := Cert_{rp}$
                let $t := s'.\mathtt{parameters}[t]$
                let $PID_{rp} := [t]Cert_{rp}.\mathtt{content}[ID_{rp}]$
                let $s'.\mathtt{parameters}[PID_{rp}] := PID_{rp}$
                let $body := \langle\langle\mathtt{PID_{rp}}, PID_{rp}\rangle\rangle$
                let $command := \langle\mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\mathtt{/reqToken}}, \mathtt{POST}, body,$
                  $\hookrightarrow$ $s'.\mathtt{refXHR}\rangle$
                let $s'.\mathtt{phase} := \mathtt{expectReqToken}$
            end if
        end if
    case $\mathtt{expectReqToken}$:
        let $pattern := \langle\mathtt{XMLHTTPREQUEST}, *, s'.\mathtt{refXHR}\rangle$
        let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
        if $input \not\equiv \bot$ then
            if $\pi_2(input) \equiv \mathtt{Unanthenticated}$ then
                let $s'.\mathtt{user} \leftarrow ids$
                let $username := s'.\mathtt{user}.name$
                let $password := \mathsf{secretOfID}(s'.\mathtt{user})$
                let $body := \langle\langle\mathtt{username}, username\rangle, \langle\mathtt{password}, password\rangle\rangle$
                let $command := \langle\mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\mathtt{/authentication}}, \mathtt{POST}, body,$
                  $\hookrightarrow$ $s'.\mathtt{refXHR}\rangle$
                let $s'.\mathtt{phase} := \mathtt{expectLoginResult}$
            else if $\pi_2(input) \equiv \mathtt{Unauthorized}$ then
                let $PID_{rp} := s'.\mathtt{parameters}[PID_{rp}]$
                let $Attr := \mathsf{GETPARAMETERS}(tree, docnonce)[\mathtt{iaKey}]$
                let $body := \langle\langle\mathtt{PID_{rp}}, PID_{rp}\rangle, \langle\mathtt{Attr}, Attr\rangle\rangle$
                let $command := \langle\mathtt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\mathtt{/authorize}}, \mathtt{POST}, body,$
                  $\hookrightarrow$ $s'.\mathtt{refXHR}\rangle$
                let $s'.\mathtt{phase} := \mathtt{expectToken}$
            else if  then
                let $IDToken := \pi_2(input)[\mathtt{IDToken}]$
                let $RPOringin := \langle s'.\mathtt{parameters}[cert].Content[\mathtt{Enpt}], \mathtt{S}\rangle$
                let $command := \langle\mathtt{POSTMESSAGE}, target, \langle\mathtt{IDToken}, IDToken\rangle, RPOrigin\rangle$
```

```
48:             let $s'$.phase := stop
49:         end if
50:     end if
51: case expectLoginResult:
52:     let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
53:     let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
54:     if $input \not\equiv \bot$ then
55:         if $\pi_2(input) \equiv \texttt{LoginSuccess}$ then
56:             let $PID_{rp} := s'.\texttt{parameters}[PID_{rp}]$
57:             let $Attr := \mathsf{GETPARAMETERS}(tree, docnonce)[\texttt{iaKey}]$
58:             let $body := \langle\langle \texttt{PID}_{\texttt{rp}}, PID_{rp}\rangle, \langle \texttt{Attr}, Attr\rangle\rangle$
59:             let $command := \langle \texttt{XMLHTTPREQUEST}, \mathsf{URL}^{IdPDomain}_{\texttt{/authorize}}, \texttt{POST}, body,$
                 $\hookrightarrow s'.\texttt{refXHR}\rangle$
60:             let $s'$.phase := expectToken
61:         end if
62:     end if
63: case expectToken:
64:     let $pattern := \langle \texttt{XMLHTTPREQUEST}, *, s'.\texttt{refXHR}\rangle$
65:     let $input := \mathsf{CHOOSEINPUT}(scriptinputs, pattern)$
66:     if $input \not\equiv \bot$ then
67:         let $IDToken := \pi_2(input)[\texttt{IDToken}]$
68:         let $RPOringin := \langle s'.\texttt{parameters}[cert].Content[\texttt{Enpt}], \texttt{S}\rangle$
69:         let $command := \langle \texttt{POSTMESSAGE}, target, \langle \texttt{IDToken}, IDToken\rangle, RPOrigin\rangle$
70:         let $s'$.phase := stop
71:     end if
72: end switch
73: stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$
```

C Proof of Security

To state the security properties for UPPRESSO, we first define an *UPPRESSO web system for authentication analysis*. This web system is based on the UPPRESSO web system and only considers one network attacker (which subsumes all web attackers and further network attackers).

Definition 9. *Let* $\mathcal{UWS}^{auth} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ *an UPPRESSO web system. We call* $\mathcal{UWS}^{auth}$ *an* UPPRESSO web system for authentication analysis *iff* $\mathcal{W}$ *contains only one network attacker process* attacker *and no other attacker processes (i.e.,* $\mathsf{Net} = \{\mathsf{attacker}\}$*,* $\mathsf{Web} = \emptyset$*).*

The security properties for UPPRESSO are formally defined as follows. First note that every *Acct* recorded in RP was calculated by RP as the result of an HTTPS `POST` request m. We refer to m as the *request corresponding to Acct*.

In the following definition, when we say a browser $b \in \mathsf{B}$ owns an *Acct*, we holds that for some relying party $rp \in \mathsf{RP}$ that calculate it and a $username \in \mathbb{S}$ with $\mathsf{ownerOfUser}(username) = b$.

$$Acct = [\mathsf{IDOfName}(username)]ID_r = [ur]G$$

We now define the similar security properties as the definition 52 in SPRESSO.

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 $r \in \mathsf{RP}$ that is honest in S^j, every RP service token of the form $\langle nonce, Acct \rangle$ recorded in $S^j(r).\mathtt{serviceTokens}$, the following two conditions are satisfied:*

(A) *If $\langle nonce, Acct \rangle$ is derivable from the attackers knowledge in S^j (i.e., $\langle nonce, Acct \rangle \in d_{\emptyset}(S^j(\mathsf{attacker}))$), then it follows that the browser b owning $Acct$ is fully corrupted in S^j (i.e., the value of isCorrupted is $\mathtt{FULLCORRUPT}$).*

(B) *If the request corresponding to $\langle nonce, Acct \rangle$ was sent by some $b \in \mathsf{B}$ which is honest in S^j, then b owns $Acct$.*

To prove Theorem 1 in section 5.1, we are going to prove the following Lemmas.

Lemma 1. *If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{UWS}^{auth}$ an honest relying party r (I) emits an HTTPS request of the form*

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and (II) in the initial state s_0 the private key k' is only known to u, and (III) u never leaks k', then all of the following statements are true:

1. *There is no state of $\mathcal{UWS}^{auth}$ where any party except for u knows k', thus no one except for u can decrypt req.*
2. *If there is a processing step $s_j \to s_{j+1}$ where the RP r leaks k to $\mathcal{W} \setminus \{u, r\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ or r is corrupted in s_j.*
3. *The value of the host header in req is the domain that is assigned the public key $\mathsf{pub}(k')$ in RP's keymapping $s_0.\mathtt{keyMapping}$ (in its initial state).*
4. *If r accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and r is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request req is not known to any atomic process p, except for the atomic DY processes r and u.*

Lemma 2. *In a run ρ of $\mathcal{UWS}^{auth}$, for every state $s_j \in \rho$, every RP $r \in \mathsf{RP}$ that is honest in s_j, every $\langle nonce, Acct \rangle \in^{\langle\rangle} S^j(r).\mathtt{serviceTokens}$, the following properties hold:*

1. *There exists exactly one $l' < j$ such that there exists a processing step in ρ of the form*

$$s_{l'} \xrightarrow[r \to \langle\langle a', f', m' \rangle\rangle]{e' \to r} s_{l'+1}$$

with e' being some events, a' and f' being addresses and m' being a service token response for Acct.

2. *There exists exactly one $l < j$ such that there exists a processing step in ρ of the form*
$$s_l \xrightarrow[r \to e]{\langle a,f,m \rangle \to r} s_{l+1}$$
with e being some events, a and f being addresses and m being a service token request for Acct.

3. *The processing steps from (1) and (2) are the same, i.e., $l = l'$.*

4. *The service token request for Acct, m in (2), is an HTTPS message of the following form:*
$$\mathsf{enc_a}(\langle\langle \mathtt{HTTPReq}, n_{req}, \mathtt{POST}, d_r, \mathtt{/authorize}, x, h, b\rangle, k\rangle, \mathsf{pub}(\mathsf{tlskey}(d_r)))$$
for $d_r \in \mathsf{dom}(r)$, some terms x, h, n_{req}, and a dictionary b such that
$$b[\mathtt{IDToken}] \equiv \langle PID_{rp}, PID_u, ver\rangle$$
with
$$PID_{rp} \equiv [S^l(r).\mathtt{loginSessions}[t]]S^l(r).\mathtt{rp},$$
$$PID_u \equiv [u]PID_{rp},$$
$$ver \equiv \mathsf{sig}(\langle PID_{rp}, PID_u\rangle, k_{sign})$$
for some nonces u, and k_{sign}.

5. *If the IdP i is honest, we have that $k_{sign} = S^l(i).\mathtt{signkey}$.*

We define the Lemma 1 and 2, which prove that the data transmitted through HTTPS is secure and the IdP's public key used for generating IDToken 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 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 ρ of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in ρ, every $r \in \mathsf{RP}$ that is honest in S^j, every RP service token of the form $\langle nonce, Acct\rangle$ recorded in $S^j(r).\mathtt{serviceTokens}$ and derivable from the attackers knowledge in S^j (i.e., $\langle nonce, Acct\rangle \in d_\emptyset(S^j(\mathsf{attacker}))$), it follows that the browser b owning Acct is fully corrupted in S^j (i.e., the value of isCorrupted is $\mathtt{FULLCORRUPT}$).*

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 exists an UPPRESSO web system* $\mathcal{UWS}^{auth}$*, a run* ρ *of* $\mathcal{UWS}^{auth}$*, a state* $s_j = (S^j, E^j, N^j)$ *in* ρ*, a RP* $r \in \mathsf{RP}$ *that is honest in* S^j*, an RP service token of the form* $\langle nonce, Acct\rangle$ *recorded in* $S^j(r)$.`serviceTokens` *and derivable from the attackers knowledge in* S^j *(i.e.,* $Acct \in d_\emptyset(S^j(\mathsf{attacker}))$*), and the browser b owning i is not fully corrupted and IdP is honest (in* S^j*).*

We now proceed to proof that this is a contradiction. First, we can see that for $\langle n, Acct\rangle$ and s_j, the conditions in Lemma 2 are fulfilled, i.e., a service token request m and a service token response m' to/from r exist, and m' is of form shown in Lemma 2 (4). Let $I := \mathsf{governor}(Acct)$. We know that I is an honest IdP. As such, it never leaks its signing key (see Algorithm 2). Therefore, the signed subterm $Content := \langle PID_{rp}, PID_u\rangle$, $ver := \mathsf{sig}(\langle PID_{rp}, PID_u\rangle, k_{sign})$ and $IDToken := \langle Content, ver\rangle$ had to be created by the IdP I. An (honest) IdP creates signatures only in Line 60 of Algorithm 2.

Lemma 3. *Under the assumption above, only the browser b can issue a request req (say,* m_{attr}*)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. We have to consider two cases for the request m_{attr}:

(A). First, if the user is not logged in with the identity u at I (i.e., the browser b has no session cookie that carries a nonce which is a session id at I for which the identitiy u is marked as being logged in, compare Line 42 of Algorithm 2), then the request has to carry (in the request body) the password matching the identity u ($\mathsf{secretOfID}(u)$) to the path `/authentication` to retrieve the session cookie. This secret is only known to b initially. Depending on the corruption status of b, we can now have two cases:

a) If b is honest in s_j, it has not sent the secret to any party except over HTTPS to I (as defined in the definition of browsers).

b) If b is close-corrupted, it has not sent it to any other party while it was honest (case a). When becoming close-corrupted, it discarded the secret.

I.e., the secret has been sent only to I over HTTPS or to nobody at all. The IdP I cannot send it to any other party. Therefore we know that only the browser b can send the request m_{attr} in this case.

(B). Second, if the user is logged in for the identity i at I, the browser provides a session id to I that refers to a logged in session at I. This session id can only be retrieved from I by logging in, i.e., case (A) applies, in particular, b has to provide the proper secret, which only itself and I know (see above). The session id is sent to b in the form of a cookie, which is set to secure (i.e., it is only sent back to I over HTTPS, and therefore not derivable by the attacker) and httpOnly (i.e., it is not accessible by any scripts). The browser b sends the cookie only to I. The IdP I never sends the session id to any other party than b. The session id therefore only leaks to b and I, and never to the attacker. Hence, the browser b is the only atomic DY process which can send the request m_{attr} in this case.

We can see that in both cases, the request was sent by b using HTTPS and I's public key: If the browser would intend to sent the request without encryption, the request would not contain the password in case (A) or the cookie in case (B). The browser always uses the "correct" encryption key for any domain (as defined in $\mathcal{UWS}^{auth}$). □

Lemma 4. *In the browser b, the request m_{attr} was triggered by script_idp loaded from the origin $\langle d, S\rangle$ for some $d \in \mathsf{dom}(I)$.*

Proof. First, $\langle d, \mathsf{S}\rangle$ for some $d \in \mathsf{dom}(I)$ is the only origin that has access to the secret $\mathsf{secretOfID}(u)$ for the identity u (as defined in Appendix B.11).

With the general properties defined in [1] and the definition of Identity Providers in Appendix B.13, in particular their property that they only send out one script, *script_idp*, we can see that this is the only script that can trigger a request containing the secret. □

Lemma 5. *In the browser b, the script script_idp receives the response to the request m_{attr} (and no other script), and at this point, the browser is still honest.*

Proof. From the definition of browser corruption, we can see that the browser b discards any information about pending requests in its state when it becomes close-corrupted, in particular any SSL keys. It can therefore not decrypt the response if it becomes close-corrupted before receiving the response.

The rest follows from the general properties defined in [1]. □

We now know that only the script *script_idp* received the response containing the IDToken. For the following lemmas, we will assume that the browser b is honest. In the other case (the browser is close-corrupted), the IA ia and any information about pending HTTPS requests (in particular, any decryption keys) would be discarded from the browser's state (as seen in the proof for Lemma 5). This would be a contradiction to the assumption (which requires that the IDToken arrived at the RP).

Lemma 6. *The script script_idp forwards the IDToken only to the script script_rp loaded from the origin $\langle d_r, \mathsf{S}\rangle$.*

Proof. It is clear that, the IDToken held by the honest *script_idp* is only sent to the origin $\langle Cert_{rp}.Enpt_{rp}, \mathsf{S}\rangle$, while the $IDToken.PID_{rp} \equiv [t]Cert_{rp}.ID_{rp}$, and t is the one-time random number. The relation of $Cert_{rp}.ID_{rp}$ and $Cert_{rp}.Enpt_{rp}$ is guaranteed by the signature $Cert_{rp}.ver$ generated by IdP I. The process is shown at Line 69 Algorithm 4. □

Lemma 7. *From the RP document, the IDToken is only sent to the RP r and over HTTPS*

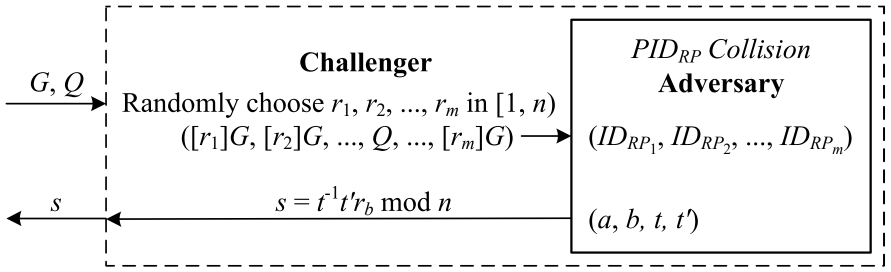
G, Q
Challenger
Randomly choose $r_1, r_2, ..., r_m$ in $[1, n)$
$([r_1]G, [r_2]G, ..., Q, ..., [r_m]G)$
PID_{RP} Collision
Adversary
$(ID_{RP_1}, ID_{RP_2}, ..., ID_{RP_m})$
(a, b, t, t')
$s = t^{-1}t'r_b \bmod n$
s

Figure 2 The PPT algorithm $\mathcal{D}_c^*$ constructed based on the PID_{RP} collision game to solve the ECDLP.

Proof. It is proved that *script_rp* of the origin $\langle Cert_{rp}.Enpt_{rp}, \mathtt{S}\rangle$ would only sent to the corresponding RP r, which is shown in Algorithm 3. □

The proofs show that the IDToken is only sent to the honest browser and target RP. It cannot be known to the attacker or any corrupted party, as none of the listed parties leak it to any corrupted party or the attacker.

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 Lemmas should be proved.

Lemma 8. *Given any two RPs in a finite set of RPs in UPPRESSO, the probability that an adversary finds different numbers t and* $t' \in [1, n)$ *satisfying* $[t]ID_{RP_j} = [t']ID_{RP_{j'}}$ *is negligible, where* $ID_{RP_j} = [r]G$*,* $ID_{RP_{j'}} = [r']G$*, r and* r' *are different numbers unknown to the adversary, and G is a generator on* $\mathbb{E}$ *of order n.*

Proof. Finding t and t' that satisfy $[t]ID_{RP_j} = [t']ID_{RP_{j'}}$, can be described as a PID_{RP}-collision game $\mathcal{G}_c$ between an adversary and a challenger: the adversary receives from the challenger a finite set of RP identities, i.e., ID_{RP_1}, ..., ID_{RP_m}, where m is the number of RPs in the system, and outputs (a, b, t, t') where $a \neq b$. If $[t]ID_{RP_a} = [t']ID_{RP_b}$, which occurs with a probability $\Pr_s$, the adversary succeeds in this game.

As depicted in Figure 2, we design a probabilistic polynomial time (PPT) algorithm $\mathcal{D}_c^*$ based on $\mathcal{G}_c$, to solve the ECDLP: find a number $x \in \mathbb{Z}_n$ satisfying $Q = [x]G$, where Q is a point on $\mathbb{E}$ and G is a generator on $\mathbb{E}$ of order n.

The algorithm $\mathcal{D}_c^*$ works as below. The input of $\mathcal{D}_c^*$ is in the form of (G, Q). On receiving an input (G, Q), the challenger first randomly chooses $r_1, \cdots, r_m$ in $\mathbb{Z}_n$ to calculate $[r_1]G, \cdots, [r_m]G$. Then, it randomly chooses $j \in [1, m]$, replaces $[r_j]G$ with Q, and sends m RP identities to the adversary, which returns the result $(a,\ b,\ t,\ t')$. Finally, the challenger calculates $s = t^{-1}t'r_b \bmod n$ and returns s as the output of $\mathcal{D}_c^*$.

If the adversary succeeds in $\mathcal{G}_c$ and $[r_a]G$ happens to be replaced with Q, then $\mathcal{D}_c^*$ outputs $s = t^{-1}t'r_b = x$ because $[tr_a]G = [t]Q = [t'r_b]G$. For the adversary, Q is indistinguishable from any other RP identities in the input set, as $[r_j]G$ is randomly replaced by the challenger. Hence, the probability of solving the ECDLP using $\mathcal{D}_c^*$ is formulated as:

$$\Pr\{\mathcal{D}_c^*(G,[x]G) = x\} = \Pr\{s = x\} = \Pr\{a = j\}\Pr_s = \frac{1}{m}\Pr_s$$

If the probability of finding t and t' satisfying $[t]ID_{RP_j} = [t']ID_{RP_{j'}}$ is non-negligible, the adversary would have advantages in $\mathcal{G}_c$ and $\Pr_s$ is non-negligible regardless of the security parameter λ. Thus, we would find that $\Pr\{\mathcal{D}_c^*(G,[x]G) = x\}$ also becomes non-negligible even when λ is sufficiently large, because m is a finite integer and $m \ll 2^\lambda$. This violates the ECDLP assumption. Therefore, the probability of finding t and t' that satisfy $[t]ID_{RP_j} = [t']ID_{RP_{j'}}$ is negligible. □

Lemma 9. *The $t^{adversary}$ is not derivable from the attackers knowledge in S^j (i.e., $t^{adversary} \in d_\emptyset(S^j(\texttt{attacker}))$), which satisfies that $IDToken.PID_{RP} \equiv [t^{adversary}]ID_{RP}^{honest}$.*

Proof. In UPPRESSO, $PID_{RP} = [t]ID_{RP}$ is generated by a user based on the target RP's identity ID_{RP} and a user-selected random number $t \in [1, n)$. Thus, PID_{RP} always specifies an RP, i.e., designates the target RP that knows t. Moreover, according to Lemma 8, given $PID_{RP} = [t]ID_{RP}$, the probability that PID_{RP} designates another RP with $ID_{RP'}$ is *negligible*. Therefore, PID_{RP} designates only the target RP with ID_{RP} in the system, so attacker cannot find a number $t^{adversary}$ □

Therefore, there is a contradication to the assumption, where we assumed that $Acct \in d_\emptyset(S^j(\mathsf{attacker}))$. 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. We say that $\mathcal{UWS}^{auth}$* is secure (with respect to Property B) *if for every run ρ of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in ρ, every $r \in \mathsf{RP}$ that is honest in S^j, every RP service token of the form Acct recorded in $S^j(r)$.*`serviceTokens`*, with the request corresponding to $\langle nonce, Acct\rangle$ sent by some $b \in \mathsf{B}$ which is honest in S^j, b owns Acct.*

First we call the request corresponding to *Acct* (or service token request) m and its response m', and we refer to the state of $\mathcal{UWS}^{auth}$ in the run ρ where r

processes m by s_l. We are going to prove the $IDToken$ uploaded by honest b can only be related with the $Acct$ owned by b.

Lemma 10. *$PID_u = [ID_u]PID_{rp}$ in IDToken uniquely identifies an account at the RP designated by PID_{rp} if and only if it receives t where $PID_{rp} = [t]ID_{rp}$ holds, and this account is uniquely mapped to a user with ID_u.*

Proof. To issue an identity token requested for PID_{rp}, the honest IdP authenticates the user with ID_u and calculates $PID_u = [ID_u]PID_{rp}$, following Equation 1. The RP designated by PID_{rp} should have received a t from the user. Following Equation 2, it can calculate $Acct = [t^{-1}]PID_u = [ID_u]ID_{rp}$, which is a *permanent* identifier determined by ID_u and ID_{rp} after the user and the RP register at the IdP. $ID_{rp} = [r]G$ is a generator on $\mathbb{E}$ of order n, as $\mathbb{E}$ is a finite cyclic group. Therefore, given a user with ID_u, $Acct$ is a *unique* point on $\mathbb{E}$ for any $u \in [1, n)$, and it is *uniquely* associated with $ID_u = u$.

$$PID_U = [ID_U]PID_{RP} = [utr]G \tag{1}$$

$$Acct = [t^{-1}utr \bmod n]G = [ur]G = [ID_U]ID_{RP} \tag{2}$$

This proves that PID_u in IDToken identifies an account $Acct$ at the designated RP, which is uniquely mapped to a user with ID_u in the system.

Next, we consider two adversarial scenarios where the attacker replays a token for another user to (1) the designated RP but receiving $t' \neq t$ in this login, and (2) any other honest RP with $ID_{rp'} = [r']G \neq [r]G$ (i.e., $r' \neq r$). In the first case, the designated RP would calculate an account as $[t'^{-1}]PID_u = [t'^{-1}ut]ID_{rp}$. Because a user's identity is randomly selected by the IdP in $\mathbb{Z}_n$ and known only to the user (and the honest IdP), the probability that $t'^{-1}ut$ happens to be the identity of another user is negligible, when n is sufficiently large. As a result, $[t'^{-1}]PID_u$ is likely not to identify any known account at the RP and therefore would be treated as a new account by the RP. Secondly, the attacker presents IDToken to RP' in the system, where $ID_{rp'} = [r']G \neq [r]G$. RP' would calculate the account as $Acct' = [\tilde{t}^{-1}]PID_u = [\tilde{t}^{-1}utrr'^{-1}][r']G = [\tilde{t}^{-1}utrr'^{-1}]ID_{rp'}$. The probability that $\tilde{t}^{-1}utrr'^{-1}$ happens to be the identity of another user at RP' is also negligible, when n is sufficiently large. □

Lemma 11. *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 $S^j(r)$.`serviceTokens`, where b owns Acct.*

Proof. Following the definiton of browser scripts, we know that m was sent by *script_rp*. The RP accepts the user's identity at line 50 in Algorithm 1. And the identity is generated at Line 48, based on the PID_u retrieved from the IDToken and the trapdoor t^{-1}. The t^{-1} is generated and set at Line 18 which is never changed. The IDToken is issued at Line 60 in Algorithm 2. The IdP generates the PID_u based on the PID_{rp} and ID_u related to $b \in \mathsf{B}$.

An attacker may allure the honest user to upload the IDToken $\in d_{\emptyset}(S^j(\texttt{attacker}))$ to honest $r \in \mathsf{RP}$, 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 18, Algorithm 4, and PID_{RP} generated at Line 20 in Algorithm 4. The t is only sent to *script_rp* at Line 9 in Algorithm 4, and the *script_rp* receives it at Line 14 in Algorithm 3. The PID_{RP} is sent to the honest IdP at Lines 58 in Algorithm 4, 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 lemma 10, the $Acct$ must be owned by honest b ($Acct \equiv [ID_U]S^j(r).ID_{RP}$, where ID_U owned by b). □

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

D Proof of Privacy against IdP-based Login Tracing

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.

We will here first describe the precise model that we use for privacy. After that, we define an equivalence relation between configurations, which we will then use in the proof of privacy.

D.1 Formal Model of UPPRESSO for Privacy Analysis

Definition 13 (Challenge Browser)**.** *Let dr some domain and $b(dr)$ a DY process. We call $b(dr)$ a* challenge browser *iff b is defined exactly the same as a browser with two exceptions: (1) the state contains one more property, namely challenge, which initially contains the term $\top$. (2) The broswer's algorithm is extended by the following at its very beginning: It is checked if a message m is addressed to the domain* `CHALLENGE` *(which we call the challenger domain). If m is addressed to this domain and no other message m' was addressed to this domain before (i.e., challenge $\not\equiv \perp$), then m is changed to be addressed to the domain dr and challenge is set to $\perp$ to recorded that a message was addressed to* `CHALLENGE`*.*

Definition 14 (Deterministic DY Process)**.** *We call a DY process $p = (I^p, Z^p, R^p, s_0^p)$* deterministic *iff the relation R^p is a (partial) function.*

We call a script R_{script} deterministic *iff the relation R_{script} is a (partial) function.*

Definition 15 (UPPRESSO Web System for Privacy Analysis)**.** *Let $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be an UPPRESSO web system with $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$,*

$\mathsf{Hon} = \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. *(as described in Appendix B.1).* $\mathsf{RP} = \{r_1, r_2\}$, *$r_1$ and r_2 two (honest) relying parties, Let* $\mathsf{attacker} \in \mathsf{Web}$ *be some web attacker. Let dr be a domain of r_1 or r_2 and $b(dr)$ a challenge browser. Let* $\mathsf{Hon}' := \{b(dr)\} \cup \mathsf{RP}$, $\mathsf{Web}' := \mathsf{Web}$*, and* $\mathsf{Net}' := \emptyset$ *(i.e., there is no network attacker). Let* $\mathcal{W}' := \mathsf{Hon}' \cup \mathsf{Web}' \cup \mathsf{Net}'$. *Let* $\mathcal{S}' := \mathcal{S}$ *and* script' *be accordingly. We call* $\mathcal{UWS}^{priv}(dr) = (\mathcal{W}', \mathcal{S}', \mathsf{script}', E^0, \mathsf{attacker})$ *an* UPPRESSO web system for privacy analysis *iff the domain dr_1 the only domain assigned to r_1, and dr_2 the only domain assigned to r_2. The browser $b(dr)$ owns exactly one identity and this identity is governed by some attacker. All honest parties (in* Hon*) are not corruptible, i.e., they ignore any* `CORRUPT` *message. Identity providers are assumed to be dishonest, and hence, are subsumed by the web attackers (which govern all identities). the relying parties already know some public key to verify UPPRESSO identity assertions from all domains known in the system and they do not have to fetch them from IdP.*

As all parties in an UPPRESSO web system for privacy analysis are either web attackers, browsers, or deterministic processes and all scripting processes are either the attacker script or deterministic, it is easy to see that in UPPRESSO web systems for privacy analysis with configuration (S, E, N) a command ζ induces at most one processing step. We further note that, under a given infinite sequence of nonces N^0, all schedules σ induce at most one run $\rho = ((S^0, E^0, N^0), \ldots, (S^i, E^i, N^i), \ldots, (S^{|\sigma|}, E^{|\sigma|}, N^{|\sigma|}))$ as all of its commands induce at most one processing step for the i-th configuration.

We will now define our privacy property for UPPRESSO:

Definition 16 (IdP-Privacy). *Let*

$$\mathcal{UWS}_1^{priv} := \mathcal{UWS}^{priv}(dr_1) = (\mathcal{W}_1, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker}_1) \text{ and}$$
$$\mathcal{UWS}_2^{priv} := \mathcal{UWS}^{priv}(dr_2) = (\mathcal{W}_2, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker}_2)$$

be UPPRESSO web systems for privacy analysis. Further, we require $\mathsf{attacker}_1 = \mathsf{attacker}_2 =: \mathsf{attacker}$ *and for $b_1 := b(dr_1)$, $b_2 := b(dr_2)$ we require $S(b_1) = S(b_2)$ and $\mathcal{W}_1 \setminus \{b_1\} = \mathcal{W}_2 \setminus \{b_2\}$ (i.e., the web systems are the same up to the parameter of the challenge browsers). We say that $\mathcal{UWS}^{priv}$ is* IdP-private *iff $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$ are indistinguishable.*

D.2 Definition of Equivalent Configurations

Let $\mathcal{UWS}_1^{priv} = (\mathcal{W}_1, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ and $\mathcal{UWS}_2^{priv} = (\mathcal{W}_2, \mathcal{S}, \mathsf{script}, E^0, \mathsf{attacker})$ be UPPRESSO web systems for privacy analysis. Let (S_1, E_1, N_1) be a configuration of $\mathcal{UWS}_1^{priv}$ and (S_2, E_2, N_2) be a configuration of $\mathcal{UWS}_2^{priv}$.

Definition 17 (Proto-Tags). *We call a term of the form $[t]R$ with the variable R as a placeholder for an ID_{rp}, and t some nonces a* proto-tag.

Definition 18 (Term Equivalence up to Proto-Tags)**.** *Let $\theta = \{a_1, \dots, a_l\}$ be a finite set of proto-tags. Let t_1 and t_2 be terms. We call t_1 and t_2* term-equivalent under a set of proto-tags θ *iff there exists a term $\tau \in \mathcal{T}_{\mathcal{N}}(\{x_1, \dots, x_l\})$ such that $t_1 = (\tau[a_1/x_1, \dots, a_l/x_l])[ID_{dr_1}/R]$ and $t_2 = (\tau[a_1/x_1, \dots, a_l/x_l])[ID_{dr_2}/R]$. We write $t_1 \rightleftharpoons_\theta t_2$.*

We say that two finite sets of terms D and D' are term-equivalent under a set of proto-tags θ *iff $|D| = |D'|$ and, given a lexicographic ordering of the elements in D of the form $(d_1, \dots, d_{|D|})$ and the elements in D' of the form $(d'_1, \dots, d_{|D'|})$, we have that for all $i \in \{1, \dots, |D|\}$: $d_i \rightleftharpoons_\theta d'_i$. We then write $D \rightleftharpoons_\theta D'$.*

Definition 19 (Equivalence of HTTP Requests)**.** *Let m_1 and m_2 be (potentially encrypted) HTTP requests, L be a set of login session tokens and $\theta = \{a_1, \dots, a_l\}$ be a finite set of proto-tags. We call m_1 and m_2* δ-equivalent under a set of proto-tags θ *iff $m_1 \rightleftharpoons_\theta m_2$ or all subterms are equal with the following exceptions:*

1. *the Host value and the Origin/Referer headers in both requests are the same except that the domain dr_1 in m_1 can be replaced by dr_2 in m_2,*
2. *If the cookie in both requests include* `loginSessionToken`*, then there exists an $l' \in L$ such that $g_1[\texttt{loginSessionToken}] \equiv l'$, and*
3. *the HTTP body g_1 of m_1 and the HTTP body g_2 of m_2 are (I) term-equivalent under θ, (II) for $j \in \{1, 2\}$ if $g_j[\texttt{IDToken}] \sim \langle PID_{dr_j}, [*]PID_{dr_j}, \mathsf{sig}(\langle PID_{dr_j}, [*]PID_{dr_j}\rangle, *)\rangle$ and the origin (HTTP header) of HTTP message in m_j is $\langle dr_j, \mathtt{S}\rangle$ then the receiver of this message is r_j, and*
4. *if m_1 is an encrypted HTTP request then and only then m_2 is an encrypted HTTP request and the keys used to encrypt the requests have to be the correct keys for dr_1 and dr_2 respectively.*

We write $m_1 \doteq_\theta m_2$.

Definition 20 (Extracting Entries from Login Sessions)**.** *Let t_1, t_2 be dictionaries over $\mathcal{N}$ and $\mathcal{T}_{\mathcal{N}}$, θ be a finite set of proto-tags, and d a domain. We call t_1 and t_2* η-equivalent *iff t_2 can be constructed from t_1 as follows: For every proto-tag $a \in \theta$, we remove the entry identified by the dictionary key i for which it holds that $\pi_2(t_1[i]) \equiv a[ID_r/R]$, if any. We denote the set of removed entries by D. We write $t_1 \trianglerighteq_r^\theta (t_2, D)$.*

Definition 21. *Let a be a proto-tag, S_1 and S_2 be states of UPPRESSO web systems for privacy analysis, and l a nonce. We call l a login session token for the proto-tag a, written $l \in \mathsf{loginSessionTokens}(a, S_1, S_2)$ iff for any $i \in \{1, 2\}$ and any $j \in \{1, 2\}$ we have that $\pi_2(S_i(r_j).\texttt{loginSessions}[l]) = a[ID_{dr_j}/R]$.*

Definition 22 (Equivalence of States)**.** *Let θ be a set of proto-tags and L be a set of login session tokens. Let $T := \{t \mid [t]R \in \theta\}$. We call S_1 and S_2* γ-equivalent under (θ, L) *iff the following conditions are met:*

1. *$S_1(\mathsf{r}_1)$ equals $S_2(\mathsf{r}_1)$ except for the subterms* `loginSessions` *and* `serviceTokens`*, and*

2. *$S_1(\mathsf{r}_2)$ equals $S_2(\mathsf{r}_2)$ except for the subterms* `loginSessions` *and* `serviceTokens`*, and*

3. *for two sets of terms D and D': $S_1(\mathsf{r}_1).\mathtt{loginSessions} \trianglerighteq^{\theta}_{dr_1} (S_2(\mathsf{r}_1).\mathtt{loginSessions}, D)$, $S_2(\mathsf{r}_2).\mathtt{loginSessions} \trianglerighteq^{\theta}_{dr_2} (S_1(\mathsf{r}_2).\mathtt{loginSessions}, D')$, and $D \rightleftharpoons_\theta D'$, and*

4. *$\forall t \in T$: $t \notin d_\emptyset(\bigcup_{i\in\{1,2\},\ A \in \mathsf{Web} \cup \mathsf{Net}\ \}} S_i(A))$*

5. *for each attacker A: $S_1(A) \rightleftharpoons_\theta S_2(A)$, and*

6. *for all $a \in \theta$ and all attackers A we have that $\nexists\ l \in \mathsf{loginSessionTokens}(a, S_1, S_2)$ such that l is a subterm of $S_1(A)$ or $S_2(A)$.*

7. *$S_1(b_1)$ equals $S_2(b_2)$ except for for the subterms* `challenge`*,* `windows` *and we have that*

   (a) *$S_1(b_1).\mathtt{challenge} = dr_1 \wedge S_2(b_2).\mathtt{challenge} = dr_2$ or $S_1(b_1).\mathtt{challenge} = S_2(b_2).\mathtt{challenge} = \bot$, and*

   (b) *$S_1(b_1).\mathtt{windows}$ equals $S_2(b_2).\mathtt{windows}$ with the exception of the subterms* `location`*,* `referrer`*,* `scriptstate`*, and* `scriptinputs` *of some document terms pointed to by $\mathsf{Docs}^+(S_1(b_1)) = \mathsf{Docs}^+(S_2(b_2)) =: J$. For all $j \in J$ we have that:*

      i. *there is no $t \in T$ such that*

      $$t \in d_{\mathcal{N}\setminus\{t\}}(\{S_1(b_1).j.\mathtt{location}, S_2(b_2).j.\mathtt{location}, \\ S_1(b_1).j.\mathtt{referrer}, S_2(b_2).j.\mathtt{referrer}\})$$

      ii. *for $p \in \{$*

      $$\langle \mathtt{XMLHTTPREQUEST}, *, * \rangle, \\ \langle \mathtt{POSTMESSAGE}, *, \langle \mathsf{dom}(dr_j), \mathtt{S}\rangle, \langle \mathtt{t}, * \rangle \rangle, \\ \langle \mathtt{POSTMESSAGE}, *, \langle \mathsf{dom}(idp), \mathtt{S}\rangle, \langle \mathtt{Cert}, * \rangle \rangle$$

      $\}$ we have $S_1(b_1).j.\mathtt{scriptinputs}|\,p \rightleftharpoons_\theta S_2(b_2).j.\mathtt{scriptinputs}|\,p$, and

      iii. *if $S_1(b_1).j.\mathtt{origin} \in \{\langle dr_1, \mathtt{S}\rangle, \langle dr_2, \mathtt{S}\rangle\}$ then $S_1(b_1).j.\mathtt{script} \equiv$* `script_rp` *and*

         A. *$S_1(b_1).j.\mathtt{location}$ and $S_2(b_2).j.\mathtt{location}$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2, and*

         B. *$S_1(b_1).j.\mathtt{referrer}$ and $S_2(b_2).j.\mathtt{referrer}$ are term-equivalent under θ except for the host part, which is either equal or dr_1 in b_1 and dr_2 in b_2, and*

        C. $S_1(b_1).j.\mathtt{scriptstate} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{scriptstate}$ *and if* $\exists\, l \in L$ *such that* l *is a subterm of* $S_1(b_1).j.\mathtt{scriptstate}$*, then* $S_1(b_1).j.\mathtt{location.host} \equiv dr_1$ *and* $S_2(b_2).j.\mathtt{location.host} \equiv dr_2$*, and*

        D. *if* $\exists\, l \in L$ *such that* l *is a subterm of* $S_1(b_1).j.\mathtt{scriptinputs}$*, then* $S_1(b_1).j.\mathtt{location.host} \equiv dr_1$ *and* $S_2(b_2).j.\mathtt{location.host} \equiv dr_2$*, and*

    iv. *if* $S_1(b_1).j.\mathtt{origin} \notin \{\langle dr_1, \mathtt{S}\rangle, \langle dr_2, \mathtt{S}\rangle\}$ *then* $S_1(b_1).j.\mathtt{script} \equiv \mathtt{script\_idp}$ *and*

        A. $S_1(b_1).j.\mathtt{location} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{location}$*, and*

        B. $S_1(b_1).j.\mathtt{referrer} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{referrer}$*, and*

        C. $S_1(b_1).j.\mathtt{scriptstate} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{scriptstate}$*, and*

        D. $S_1(b_1).j.\mathtt{scriptinputs} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{scriptinputs}$*, and*

        E. $\forall t \in T$*:* t *is not contained in any subterm of* $S_1(b_1).j.\mathtt{scriptstate}$ *except for* $S_1(b_1).j.\mathtt{scriptstate}.\mathit{parameters}[\mathtt{t}]$*, and*

        F. $\nexists\, l \in L$ *such that* l *is a subterm of* $S_1(b_1).j.\mathtt{scriptstate}$ *or of* $S_1(b_1).j.\mathtt{scriptinputs}$*, and*

  (c) *for* $x \in \{\mathtt{cookies}, \mathtt{localStorage}, \mathtt{sessionStorage}, \mathtt{sts}\}$ *we have that* $S_1(b_1).x \rightleftharpoons_\theta S_2(b_2).x$*. For the domains* dr_1 *and* dr_2 *there are no entries in the subterms* x*.*

Definition 23 (Equivalence of Events)**.** *Let* θ *be a set of proto-tags,* L *be a set of login session tokens,* H *be a set of nonces, and* $T := \{t \mid [t]R \in \theta\}$*. We call* $E_1 = (e^{(1)}_1, e^{(1)}_2 \dots)$ *and* $E_2 = (e^{(2)}_1, e^{(2)}_2 \dots)$ β-equivalent under (θ, L, H) *iff all of the following conditions are satisfied for every* $i \in \mathbb{N}$*:*

1. *One of the following conditions holds true:*

   (a) $e^{(1)}_i \rightleftharpoons_\theta e^{(2)}_i$ *and if* $e^{(1)}_i$ *contains an HTTP(S) message (i.e., HTTP(S) request or HTTP(S) response), then the HTTP nonce of this HTTP(S) message is not contained in* H*, or*

   (b) $e^{(1)}_i$ *is an HTTP request* m_1 *from* b_1 *to* r_1 *and* $e^{(2)}_i$ *is an HTTP request* m_2 *from* b_2 *to* r_2*,* $m_1 \risingdotseq_\theta m_2$*, and both requests are unencrypted or encrypted (i.e.,* m_1 *and* m_2 *are the content of the encryption) and* $m_1.\mathtt{nonce} \in H$*, or*

   (c) $e^{(1)}_i$ *is an HTTP(S) response from* r_1 *to* b_1 *and* $e^{(2)}_i$ *is an HTTP(S) response from* r_2 *to* b_2*, and their HTTP messages* m_1 *(contained in* $e^{(1)}_i$*) and* m_2 *(contained in* $e^{(1)}_i$*) are the same except for the HTTP body* $g_1 := m_1.\mathtt{body}$ *and the HTTP body* $g_2 := m_2.\mathtt{body}$ *which have to be* $g_1 \rightleftharpoons_\theta g_2$ *and* $m_1.\mathtt{nonce} \in H$*.*

2. *If there exists* $l \in L$ *such that* l *is a subterm of* $e^{(1)}_i$ *or* $e^{(2)}_i$ *then we have that* $e^{(1)}_i$ *is a message from* b_1 *to* r_1 *and* $e^{(2)}_i$ *is a message from* b_2 *to* r_2 *or*

we have that $e_i^{(1)}$ is a message from r_1 to b_1 and $e_i^{(2)}$ is a message from r_2 to b_2.

3. *If there exists $t \in T$ such that $t \in d_{\mathcal{N}\setminus\{t\}}(\{e_i^{(1)}, e_i^{(2)}\})$ then $e_i^{(1)}$ is an HTTP(S) request from b_1 to r_1 and $e_i^{q(2)}$ is an HTTP(S) request from b_2 to r_2 and the bodies of both HTTP messages are of the form $\langle\langle \mathtt{t}, t\rangle\rangle$.*

4. *If $e_i^{(1)}$ or $e_i^{(2)}$ is an HTTP(S) response with body g from a relying party, then it does not contain any* `Location` *or* `Strict-Transport-Security` *header and if $\pi_1(g)$ is a string representing a script, then $\pi_1(g)$ is* `script_rp`.

5. *If $e_i^{(1)}$ or $e_i^{(2)}$ is an unencrypted HTTP response, then the message was sent by some attacker.*

Definition 24 (Equivalence of Configurations)**.** *We call (S_1, E_1, N_1) and (S_2, E_2, N_2) α-*equivalent *iff there exists a set of proto-tags θ and a set of nonces H such that S_1 and S_2 are γ-equivalent under (θ, H), E_1 and E_2 are β-equivalent under (θ, L, H) for $L := \bigcup_{a\in\theta} \mathsf{loginSessionTokens}(a, S_1, S_2)$, and $N_1 = N_2$.*

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

Proof. From the α-equivalence of (S_1, E_1, N_1) and (S_2, E_2, N_2) it follows that $S_1(\mathsf{attacker}) \rightleftharpoons_\theta S_2(\mathsf{attacker})$. From Condition 4 for γ-equivalence it follows that $t \notin d_\emptyset(\bigcup_{i\in\{1,2\},\ A\in \mathsf{Web}\cup\mathsf{Net}} S_i(A))$ (i.e., the attacker does not know any keys for the tags contained in its view), and therefore it is easy to see that the views are statically equivalent. □

We now show that $\sigma(\mathcal{UWS}_1^{priv}) \approx \sigma(\mathcal{UWS}_2^{priv})$ by induction over the length of σ. We first, in Lemma 13, show that α-equivalence (and therefore, indistinguishability of the views of $\mathsf{attacker}$) holds for the initial configurations of $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$. We then, in Lemma 14, show that for each configuration induced by a processing step in ζ, α-equivalence still holds true.

Lemma 13. *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. We now have to show that there exists a set of proto-tags θ and a set of nonces H such that S_1^0 and S_2^0 are γ-equivalent under (θ, H), $E_1^0 = E^0$ and $E_2^0 = E^0$ are β-equivalent under (θ, L, H) with $L := \bigcup_{a\in\theta} \mathsf{loginSessionTokens}(a, S_1, S_2)$, and $N_1^0 = N_2^0 = N^0$.

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 $S_1^0(b_1)$ and $S_2^0(b_2)$ are equal. Therefore, all conditions of Definition 22 are fulfilled. Hence, the initial configurations are α-equivalent. □

Lemma 14. *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 latter case, let (S_1', E_1', N_1') and (S_2', E_2', N_2') be configurations induced by ζ such that*

$$(S_1, E_1, N_1) \xrightarrow{\zeta} (S_1', E_1', N_1') \quad \textit{and} \quad (S_2, E_2, N_2) \xrightarrow{\zeta} (S_2', E_2', N_2') \ .$$

Then, (S_1', E_1', N_1') and (S_2', E_2', N_2') 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 $L := \bigcup_{a\in\theta} \text{loginSessionTokens}(a, S_1, S_2)$, $T := \{t \mid [t]R \in \theta\}$.

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

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

<u>*Case $p_1 = r_1$:*</u> 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 10;
- path=/`loginSSO`(start a login), Line 13;
- path=/`startNegotiation`(derive a PID_{rp}), Line 16;
- path=/`uploadToken`(verify ID token, calculate Acct), Line 27.

From the cases in Definition 23, only two can possibly apply here:Case 1a and Case 1b. 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
$$\langle \mathit{HTTPResp}, n, 200, \langle\rangle, \mathit{RPScript}\rangle \qquad (4)$$
We can note that Condition 4 of Definition 23 holds true and.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`

- path=/`uploadToken`

Definition 23, Case 1b: $e_i^{(1)}$ is an HTTP(S) request from b_1 to r_1 and $e_i^{(2)}$ is an HTTP(S) request from b_2 to r_2. This case implies $p_2 = \mathsf{r}_2$.
We note that Condition 4 of Definition 23 holds for the same reasons as in the previous case. As the response is always addressed to the IP address of b_1 or b_2, respectively, Condition 4 of Definition 23 is fulfilled.
As we see below, for the output events $E_{\text{out}}^{(1)}$ and $E_{\text{out}}^{(2)}$ (if any) only Case 1c of Definition 23 applies. This implies that the output events must contain an HTTP nonce contained in H. As we know that the HTTP nonce of the incoming HTTP requests is contained in H and the output HTTP responses (if any) of the RP reuses the same HTTP nonce, the nonce of the HTTP responses is in H.

- *path* = /`script` In this case, the output events contain no $l \in L$ or $t \in T$ meaning that E_1' and E_2' being β-equivalent under (θ, H, L) according to Definition 23, Case 1c. As there are no changes to any state, we have that S_1' and S_2' are γ-equivalent under (θ, H). No new nonces are chosen, hence, $N_1 = N_1' = N_2 = N_2'$.

- *path* = /`loginSSO` This case is analogue to the case above.

- *path* = /`startNegotiation` In this case, an HTTP response is created. We denote the HTTP response generated by r_1 as m_1' and the one generated by r_2 as m'_2. We then have that

$$m'_1 = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, g_1\rangle, k)$$
$$m'_2 = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, g_2\rangle, k)$$

with

$$g_1 = \langle\langle \mathtt{Cert_{RP}}, S_1(r_1).\mathtt{IdPConfig}.Cert_{RP}\rangle, \langle \mathtt{loginSessionToken}, \nu_1\rangle\rangle$$
$$g_2 = \langle\langle \mathtt{Cert_{RP}}, S_2(r_2).\mathtt{IdPConfig}.Cert_{RP}\rangle, \langle \mathtt{loginSessionToken}, \nu_1\rangle\rangle$$

Obviously, m'_1 equals m'_2. For $N_1 = N_2 = (n_1, n_2, \dots)$, We set $\theta' = \theta \cup \{[t]S_j(r_j).ID_{RP}\}$ for $j \in \{1,2\}$, $N'_1 = N'_2 = (n_2, \dots)$ (as exactly one nonce is chosen in both processing steps) and $L' = L \cup \{n_1\}$. The receiver of both messages is the browser b_1 or b_2, respectively. Obviously, it holds that $L' = \bigcup_{a\in\theta'} \mathsf{loginSessionTokens}(a, S'_1, S'_2)$ and there exists an $l' \in L'$ such that $g_1[\mathtt{loginSessionToken}] \equiv l'$. As Conditions 1c and 3 of Definition 23 hold, E'_1 and E'_2 are β-equivalent under (θ', H, L'). The subterm `loginSessions` of $S_1(r_1)$ is extended exactly the same as the subterm `loginSessions` of $S_2(r_2)$. Thus, we have that S'_1 and S'_2 are γ-equivalent under (θ', H).

- *path* $=$ /`uploadToken` In this case, there are four checks at Algorithm 1 in Line 30, 41, 41 and 45.

  From Condition 3 of Definition 22 we know that for $ls_1 := S_1(r_1).\mathtt{loginSessions}[l]$ and $ls_2 := S_2(r_2).\mathtt{loginSessions}[l]$, we have that $ls_1 \rightleftharpoons_\theta ls_2$. Therefore, we have that if the first two checks fail in r_1 then and only then they fail in r_2.

  As we know that $m_1 \doteq_\theta m_2$, we have that if the third check fails in r_1 then and only then it fails in r_2. The same holds true for the fourth check.

  if r_1 and r_2 both accept the IDToken, they will generate HTTP responses with service Token.We denote the HTTP response generated by r_1 as m'_1 and the one generated by r_2 as m'_2. We then have that

$$m'_1 = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, g_1\rangle, k)$$
$$m'_2 = \mathsf{enc_s}(\langle \mathtt{HTTPResp}, n, 200, \langle\rangle, g_2\rangle, k)$$

  with

$$g_1 = \langle\langle \mathtt{nonce}, \nu_1\rangle\rangle$$
$$g_2 = \langle\langle \mathtt{nonce}, \nu_1\rangle\rangle$$

  Same as above, m'_1 equals m'_2, $N'_1 = N'_2 = (n_2, \dots)$ and $L' = L$. The receiver of both messages is the browser b_1 or b_2, respectively. As Conditions 1c and 2 of Definition 23 hold, E'_1 and E'_2 are β-equivalent under (θ, H, L). The subterm `loginSessions` of $S_1(r_1)$ is extended exactly the same as the subterm `loginSessions` of $S_2(r_2)$. Thus, we have that S'_1 and S'_2 are γ-equivalent under (θ, H).

Case $p_1 = \mathsf{r}_2$: This case is analogue to the case $p_1 = \mathsf{r}_1$ above. Note that the Case 1b of Definition 23 cannot occur by definition.

Case $p_1 = \mathsf{b}_1$: $\implies p_2 = \mathsf{b}_2$

TRIGGER We now distinguish between the possible values for cmd_{switch}.

1 (trigger script): In this case, the script in the window indexed by cmd_{window} is triggered. Let j be a pointer to that window.

We first note that such a window exists in b_1 iff it exists in b_2 and that $S_1(b_1).j.\mathtt{script} \equiv S_2(b_2).j.\mathtt{script}$. We now distinguish between the following cases, which cover all possible states of the windows/documents:

1. $S_1(b_1).j.\mathtt{origin} \in \{\langle dr_1, \mathtt{S}\rangle, \langle dr_2, \mathtt{S}\rangle\}$ and $S_1(b_1).j.\mathtt{script} \equiv \mathtt{script\_rp}$.

   Similar to the following scripts, the main distinction in this script is between the script's internal states (named `phase`). With the term-equivalence under proto-tags θ we have that either $S_1(b_1).j.\mathtt{scriptstate.phase} = S_2(b_2).j.\mathtt{scriptstate.phase}$ or the script's state contains a tag and is therefore in an illegal state (in which case the script will stop without producing output or changing its state).

   We can therefore now distinguish between the possible values of $S_1(b_1).j.\mathtt{scriptstate.phase} = S_2(b_2).j.\mathtt{scriptstate.phase}$:

   start: In this case, the script open a blank page addressed to its own origin, which is either (a) equal and $\langle dr_1, \mathtt{S}\rangle$ or $\langle dr_2, \mathtt{S}\rangle$ or it is (b) $\langle dr_1, \mathtt{S}\rangle$ in b_1 and $\langle dr_2, \mathtt{S}\rangle$ in b_2. The path is the (static) string `/loginSSO`. The script saves a (static) value for `phase` in its scriptstate.

   In both Cases, we have that the command is term-equivalent under proto-tags θ and hence, the browser emits a HTTP request which is term-equivalent.Hence, we have γ-equivalence under (θ, H) for the new states, β-equivalence under (θ, H, L) for the new events, and α-equivalence for the new configuration.

   expectt: In this case, the script retrieves the result of a postMessage from *scriptinputs*. As we know that $S_1(b_1).j.\mathtt{scriptstate} \rightleftharpoons_\theta S_2(b_2).j.\mathtt{scriptstate}$ and that for all matching postMessages that they also have to be term-equivalent up to θ and that the window structure is equal in both browsers, we have that either the same postMessage is retrieved from *scriptinputs* or none in both browsers.

   Then the script saves a (static) value for `phase` in its scriptstate, and we set $H' := H \cup \{n\}$ with n being the nonce that the browser chooses for λ_1. Therefore, we have γ-equivalence under (θ, H') for the new states. We also have β-equivalence under (θ, H', L) for the new events, and α-equivalence for the new configuration.

expectCert: In this case, the script retrieves the result of an XHR from *scriptinputs* that matches the reference contained in *scriptstate*. From Condition 7(b)iii of Definition 22 we know that all results from XHRs in *scriptinput* are term-equivalent up to θ and that *scriptstate* is term-equivalent up to θ. Hence, in both browsers, both scripts stop with an empty command or both continue as they successfully retrieved such an XHR.

The script now constructs a postMessage that is sent to exactly the same window in both browsers and that requires that the receiver origin has to be $\langle \mathtt{IdPdomain}, \mathtt{S} \rangle$ The postMessage is only sent to this origin, we have that γ-equivalence cannot be violated.

We now have that S'_1 and S'_2 are γ-equivalent under (θ, H), E'_1 and E'_2 are β-equivalent under (θ, H, L), and as exactly none of nonces is chosen, we have that the new configuration is α-equivalent.

expectToken: This case is the same as `expectt` and we have that the new configuration is α-equivalent.

2. $S_1(b_1).j.\mathtt{origin} \notin \{\langle dr_1, \mathtt{S}\rangle, \langle dr_2, \mathtt{S}\rangle\}$. $S_1(b_1).j.\mathtt{script} \equiv \mathtt{script\_idp}$.

Unlike SPRESSO, $\mathtt{script}_{\mathtt{idp}}$ is trustful in UPPRESSO due to the use of SRI (Subresource Integrity). Because of this check, browsers can control the content of $\mathtt{script}_{\mathtt{idp}}$ downloaded from the Identity Provider. Hence, we now analyze every internal state just like in $\mathtt{script}_{\mathtt{rp}}$.

start: In this case, the script chooses a new nonce for t. Since t is only stored in $S_1(b_1).j.\mathtt{scriptstate}.\mathit{parameters}$, the condition 4 and condition 7(b)ivE of Definiton 22 hold. Hence, we have γ-equivalence under (θ, H) for the new states.

From the equivalence definition of states (Definition 22) we can see that the window tree has the same structure in both processing steps. So the script now constructs a postMessage that is sent to exactly the same window in both browsers and that requires that the receiver has to be the opener of this window. Since the new tag hasn't been generated and Condition 1a of Definition 23 holds, we have β-equivalence under (θ, H, L).

expectCert: The same as above, we can have that either the same postMessage is retrieved from *scriptinputs* or none in both browsers and the result of *checksig* is same as well. The state $Cert_{rp}$ is equal in both `scriptstate` and the state PID_{rp} is term-equivalent under θ. As the condition 7(b)ivC

of Defition 22 holds, we have γ-equivalence under (θ, L) for the new states.
Obviously, we have β-equivalence under (θ, H, L) as Condition 1a of Definition 23 holds.

expectReqToken: In this case, the script retrieves the result of an XHR from *scriptinputs* that matches the reference contained in *scriptstate*. From Condition 7(b)iii of Definition 22 we know that all results from XHRs in *scriptinput* are term-equivalent up to θ and that *scriptstate* is term-equivalent up to θ. Hence, in both browsers, both scripts will reach the same if-else branch.
Since there aren't any new states stored and the requests' destination are fixed, we can have γ-equivalence under (θ, L) for the new states and β-equivalence under (θ, H, L).

expectLoginResult: This case is the same as the second branch of `expectReqToken`.

expectToken: This case is the same as the third branch of `expectReqToken`.

3. $S_1(b_1).j.\mathtt{origin} \not\in \{\langle dr_1, \mathtt{S}\rangle, \langle dr_2, \mathtt{S}\rangle\}$. $S_1(b_1).j.\mathtt{script} \not\equiv \mathtt{script\_idp}$.
Here, we assumte that the script in this case is the the attacker script R^{att}, as it subsumes all other scripts.
We first observe, that its "view", i.e., the input terms it gets from the browser, is term-equivalent up to proto-tags θ between (the scripts running in) $S_1(b_1)$ and $S_2(b_2)$. From the equivalence definition of states (Definition 22) we can see that:
   - The window tree has the same structure in both processing steps. All window terms are equal (up to their `documents` subterm). All same-origin documents contain only subterms that are term-equivalent up to θ (again, up to their `subwindows` subterms). All non-same-origin documents become limited documents and therefore are equal (up to the subwindows, limited documents only contain the subwindows and the document nonce).
   - The subterms `cookies`, `localStorage`, `sessionStorage`, `scriptstate`, and `scriptinputs` are term-equivalent up to θ.
   - The subterms `ids` and `secrets` are equal.
   - There is not $t \in T$ as a subterm (except as keys for tags) in this view. We therefore have that no such term can be contained in the output command of the script, or in the new scriptstate.

As the input of the script as a whole is term-equivalent up to θ, does not contain any placeholders in V_{script}, and does not contain

a key for any tag in θ, we have that the output of the script, i.e., *scriptstate*$'$, *cookies*$'$, *localStorage*$'$, *sessionStorage*$'$, *command*$'$, must be term-equivalent up to proto-tags θ (in particular, the same number of nonces is replaced in both output terms in both processing steps). Note that the first element of the command output must be equal between the two browsers (as it must be string) or otherwise the browsers will ignore the command in both processing steps.
Analogously, we see that the input does not contain any subterm $l \in L$.
We can now distinguish the possible commands the script can output (again, all parameters for these commands must be term-equivalent under θ):

(a) Empty or invalid command: In this case, the browser outputs no message and its state is not changed. α-equivalence is therefore trivially given.

(b) $\langle \texttt{HREF}, \mathit{url}, \mathit{hrefwindow}, \mathit{noreferrer}\rangle$: Here, the browser calls GETNAVIGABLEWINDOW to determine the window in which the document will be loaded. Due to the synchronous window structure between the two browsers, the result will be the same in both processing steps (which may include creating a new window with a new nonce).
Now, a new HTTP(S) request is assembled from the URL. A Referer header is added to the request from the document's current `location` (which is term-equivalent under θ) and given to the SEND function. There, if the `host` part of the URL is `CHALLENGE`, it will be replaced by dr_1 in b_1 and by dr_2 in b_2. (In this case, the α-equivalence in the following holds for $H' := H \cup \{n\}$, where n is the nonce of the generated HTTP request. Otherwise, it holds for $H' := H$.). Afterwards, for domains that are in the `sts` subterm of the browser's state, the request will be rewritten to HTTPS. Any cookies for the domain in the requests are added. Note that both latter steps never apply to requests to dr_1 or dr_2 as per definition, there are no entries for these domains in *sts* and *cookies*. The same number of nonce is chosen in both processing steps, and therefore α-equivalence holds.

(c) $\langle \texttt{IFRAME}, \mathit{url}, \mathit{window}\rangle$ This case is completely parallel to Case 3b.

(d) $\langle \texttt{FORM}, \mathit{url}, \mathit{method}, \mathit{data}, \mathit{hrefwindow}\rangle$ This case is parallel to Case 3b, except that an Origin header is added. Its properties are the same as those of the Referer header in Case 3b.

(e) $\langle \texttt{SETSCRIPT}, \mathit{window}, \mathit{script}\rangle$ In this case, the same document is manipulated in both processing steps in the same way. Note that only same-origin documents, i.e., attacker documents, can be manipulated. No output event is generated, and no nonces are chosen. α-equivalence is given trivially.

(f) $\langle$SETSCRIPTSTATE, *window*, *scriptstate*$\rangle$ This case is parallel to Case 3e.

(g) $\langle$XMLHTTPREQUEST, *url*, *method*, *data*, *xhrreference*$\rangle$ This case is parallel to Case 3b with the addition of the Origin header (see Case 3d) and the addition of a reference parameter. Therefore, for γ-equivalence, it is important to note that this reference can only be a nonce (and therefore is equal in both processing steps). Otherwise, the browser stops in both processing steps.

(h) $\langle$BACK, *window*$\rangle$, $\langle$FORWARD, *window*$\rangle$, and $\langle$CLOSE, *window*$\rangle$ If the script outputs one of these commands, in both processing steps, the browsers will be manipulated in exactly the same way. No output events are generated, and no nonces are chosen.

(i) $\langle$POSTMESSAGE, *window*, *message*, *origin*$\rangle$ In this case, a term containing *message* (term-equivalent under θ) is added to a document's `scriptinput` term. If the *origin* is $\perp$, the same term will be added to the same document in both processing steps. Otherwise, the term may only be added to one document (if, for example, the origin is $\langle dr_1, \mathtt{S}\rangle$ and the target documents in both browsers have the domain dr_1 and dr_2, respectively). In this case, however, the equivalence defined on the scriptinputs is preserved.

2 (navigate to URL): In this case, a new window is opened in the browser and a document is loaded from *url*.

The states of both browsers are changed in the same way except if the domain of the URL is CHALLENGE. In both cases, a new (at this point empty) window is created and appended the `windows` subterm of the browsers. This subterm is therefore changed in exactly the same way.

A new HTTP request is created and generated requests in both processing steps can only differ in the host part iff the domain is CHALLENGE. In this case, in b_1 the domain is replaced by dr_1 and in b_2 by dr_2 and the α-equivalence in the following holds for $H' := H\{n\}$, where n is the nonce of the generated HTTP request.

The request cannot contain any $l \in L$ or $t \in T$. and

the Condition 1a of Definition 23.

In both processing steps, three nonces are chosen.

Therefore, we have α-equivalence for (S'_1, E'_1, N'_1) and (S'_2, E'_2, N'_2).

3 (reload document): Here, an existing document is retrieved from its original location again. From the definition of γ-equivalence under (θ, L) we can see that whatever document is reloaded, its location is either (I) term-equivalent under θ, or (II) it is term-equivalent under θ except for the domain, which is dr_1 in b_1 and dr_2 in b_2.

We note that in either case, the requests are constructed from the location and referrer properties of the document that is to be reloaded, and therefore, cannot contain any $t \in T$.

In Case (I), we note that the domain cannot be `CHALLENGE`. If the document is reloaded, the same request is issued in both browsers (therefore, β-equivalence under (θ, H, L) is given), and none states are changed such that we have γ-equivalence under (θ, L). The same number of nonces is chosen in both runs, and we have α-equivalence.

Case (II) is similar, but we have $H' := H \cup \{n\}$, where n is the nonce of the HTTP request. Then we have β-equivalence under (θ, H', L). Again, the same number of nonces is chosen and we have α-equivalence.

Other Any other message is discarded by the browsers without any change to state or output events.

Lemma 15. *In UPPRESSO, the IdP cannot distinguish $PID_{RP} = [t]ID_{RP}$ from a random variable on $\mathbb{E}$, where t is random in $\mathbb{Z}_n$.*

Proof. Consider a finite cyclic group $\mathbb{E}$ where the number of points on $\mathbb{E}$ is n. Because G is a generator of order n, $ID_{RP} = [r]G$ is also a generator on $\mathbb{E}$ of order n. t is randomly chosen in $\mathbb{Z}_n$ and always kept unknown to the IdP. Therefore, $PID_{RP} = [t]ID_{RP}$ is *indistinguishable* from a point Q that is randomly chosen on $\mathbb{E}$. □

Case p_1 is some attacker:

Here, only Case 1a from Definition 23 can apply to the input events, i.e., the input events are term-equivalent under proto-tags θ. This implies that the message was delivered to the same attacker process in both processing steps. Let A be that attacker process. With Case 5 of Definition 22 we have that $S_1(A) \rightleftharpoons_\theta S_2(A)$ and with Case 3 and Case 4 of Definition 23 and lemma 15 it follows immediately that the attacker cannot decrypt any of the tags in θ in its knowledge. Further, in the attackers state there are no variables (from V_{process}).

With the output term being a fixed term (with variables) $\tau_{\text{process}} \in \mathcal{T}_{\mathcal{N}}(\{x\} \cup V_{\text{process}})$ and x being $S_1(A)$ or $S_2(A)$, respectively, and there is no subterm $l \in L$ contained in either $S_1(A)$ or $S_2(A)$ (Condition 6 of Definition 22), it is easy to see that the output events are β-equivalent under θ, i.e., $E^{(1)}_{\text{out}} \rightleftharpoons_\theta E^{(2)}_{\text{out}}$, there are not $t \in T$ contained in the output events and the used nonces are the same, i.e., $N'_1 = N'_2$. The new state of the attacker in both processing steps consists of the input events, the output events, and the former state of the event, and, as such, is β-equivalent under proto-tags θ. Therefore we have α-equivalence on the new configurations. □

This proves Theorem 1. ■

E Proof of Privacy against RP-based Identity Linkage

E.1 Formal Model of UPPRESSO for Privacy Analysis

Definition 25 (Challenge IdP)**.** *Let dr some domain and $idp(\langle dr_1, dr_2, u\rangle)$ a DY process. We call it a* challenge IdP *iff b is defined exactly the same as a identity server with two exceptions: (1) the state contains one more property, namely challenge, which initially contains the term $\top$. (2) The IdP's algorithm is modified by the following at line 40 in algorithm 2: It is checked if the login request m is addressed to the domain dr_1 If m is addressed to this domain, then the PID_u is generated using the given u. It is also checked if the login request m is addressed to the domain dr_2 and no other message m' was addressed to this domain before (i.e., challenge $\not\equiv \perp$), then the PID_u is generated using the given u and challenge is set to $\perp$ to recorded that a message was addressed to dr_2.*

Definition 26 (UPPRESSO Web System for Privacy Analysis)**.** *Let $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be an UPPRESSO web system with $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Web} \cup \mathsf{Net}$, $\mathsf{Hon} = \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. (as described in Appendix B.1). $\mathsf{B} = \{b_1, b_2\}$, b_1 is honest and b_2 is malicious and they both own some identities. $\mathsf{RP} = \{r_1, r_2\}$, r_1 and r_2 two (malicious) relying parties, Let $\mathsf{attacker} = \{b_2\} \cup \{r_1, r_2\}$ be some web attacker. Let dr_1 be the domain of r_1, dr_2 be the domain of r_2 and u_{idp} be an identity owned only by IdP, then $idp_c = idp(\langle dr_1, dr_2, u_{idp}\rangle)$ is a challenge IdP. Let $\mathsf{Hon}' := \mathsf{B} \cup \{idp_c\}$, $\mathsf{Web}' := \mathsf{Web}$, and $\mathsf{Net}' := \emptyset$ (i.e., there is no network attacker). Let $\mathcal{W}' := \mathsf{Hon}' \cup \mathsf{Web}' \cup \mathsf{Net}'$. Let $\mathcal{S}' := \mathcal{S} \setminus \{\mathtt{script\_rp}\}$ and script' be accordingly. We call $\mathcal{UWS}^{priv}(dr_1, dr_2, u_{idp}) = (\mathcal{W}', \mathcal{S}', \mathsf{script}', E^0, \mathsf{attacker})$ an* UPPRESSO web system for privacy analysis *iff the domain dr_1 the only domain assigned to r_1, and dr_2 the only domain assigned to r_2. All honest parties (in Hon) are not corruptible, i.e., they ignore any `CORRUPT` message. Relying Parties and some browsers are assumed to be dishonest, and hence, are subsumed by the web attackers.*

References

[1] D. Fett, R. Küsters, and G. Schmitz. Analyzing the browserid sso system with primary identity providers using an expressive model of the web. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I 20*, pages 43–65. Springer, 2015. C.1, C.1