1 Algorithms

1 Algorithm idp

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

let $s' := s$
let $n, method, path, parameters, headers, body$ **such that** $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$ **if** $possible$; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{IdPScript}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /authentication$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $username := body[username]$
  let $password := body[password]$
  if $password \not\equiv \texttt{PasswordOfUser}(\texttt{username})$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginFailure}\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $session[uid] := \texttt{UIDOfUser}(username)$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{LoginSucess}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /reqToken$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $IDTokens := session[IDTokens]$
  if $IDTokens[body[PID_{RP}]] \not\equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, IDTokens[body[PID_{RP}]]\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Unauthenticated}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /authorize$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $uid := session[uid]$
  if $uid \equiv \texttt{null}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if
  let $PID_{RP} := parameters[PID_{RP}]$
  if $\texttt{IsValid}(PID_{RP}) \equiv \texttt{FALSE}$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    stop $\langle b, a, m'\rangle, s'$

end if
if $\texttt{IsInScope}(uid, body[Attr]) \equiv \texttt{FALSE}$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
  stop $\langle b, a, m'\rangle, s'$
end if
let $ID_U := session[uid]$
let $PID_U := \texttt{Multiply}(PID_{RP}, ID_U)$
let $Validity := \texttt{CurrentTime}() + s'.Validity$
let $Content := \langle PID_{RP}, PID_U, s'.Issuer, Validity\rangle$
let $Sig := \texttt{SigSign}(Content, s'.SK)$
let $IDToken := \langle Content, Sig\rangle$
let $session[IDTokens] := session[IDTokens] +^{\langle\rangle} \langle PID_{RP}, IDToken\rangle$
let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, Token\rangle$
stop $\langle b, a, m'\rangle, s'$
end if
stop $\langle\rangle, s'$

2 Algorithm rp

Input: $\langle a, b, m\rangle, s$
let $s' := s$
let $n, method, path, parameters, headers, body$ **such that** $\langle \texttt{HTTPReq}, n, method, path, parameters, headers, body\rangle \equiv m$ **if** $possible$; **otherwise** stop $\langle\rangle, s'$
if $path \equiv /script$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{RPScript}\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /loginSSO$ **then**
  let $m' := \langle \texttt{HTTPResp}, n, 302, \langle\langle \texttt{Location}, s'.IdP.ScriptUrl\rangle\rangle, \langle\rangle\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /startNegotiation$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  let $t := body[t]$
  let $t^{-1} := \texttt{Inverse}(t)$
  let $session[t^{-1}] := t^{-1}$
  let $session[state] := expectToken$
  let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{Cert}_{\texttt{RP}}, s'.Cert_{RP}\rangle\rangle$
  stop $\langle b, a, m'\rangle, s'$
else if $path \equiv /uploadToken$ **then**
  let $cookie := headers[Cookie]$
  let $session := s'.SessionList[cookie]$
  if $session[state] \not\equiv expectToken$ **then**
    let $m' := \langle \texttt{HTTPResp}, n, 200, \langle\rangle, \texttt{Fail}\rangle$
    stop $\langle b, a, m'\rangle, s'$
  end if

```
    let IDToken := body[IDToken]
    if checksig(IDToken.Content, IDToken.Sig, s'.IdP.PK) ≡ FALSE then
        let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨b, a, m'⟩, s'
    end if
    let Time := CurrentTime()
    let Content := Token.Content
    if Time > Content.Validity then
        let m' := ⟨HTTPResp, n, 200, ⟨⟩, Fail⟩
        stop ⟨b, a, m'⟩, s'
    end if
    let PID_U := Content.PID_U
    let t^{-1} := session[t^{-1}]
    let Acct := Multiply(PID_U, t^{-1})
    if Acct ∉ ListOfUser() then
        let AddUser(Acct)
    end if
    let session[user] := Acct
    let s'.serviceTokens := s'.serviceTokens +^{⟨⟩} ⟨IDToken, Acct⟩
    let m' := ⟨HTTPResp, n, 200, ⟨⟩, LoginSuccess⟩
    stop ⟨b, a, m'⟩, s'
end if
stop ⟨⟩, s'
```

3 Algorithm script_idp

```
Input: ⟨tree, docID, scriptstate, scriptinputs, cookies, ids, secret⟩
let s' := scriptstate
let command := ⟨⟩
let target := PARENTWINDOW(tree, docID)
let IdPDomain := s'.IdPDomain
switch s'.phsae do
    case start:
        let t := Random()
        let command := ⟨POSTMESSAGE, target, ⟨t, t⟩, null⟩
        let s'.Parameters[t] := t
        let s'.phase := expectCert
    case expectCert:
        let pattern := ⟨POSTMESSAGE, target, *, ⟨Cert_RP, *⟩⟩
        let input := CHOOSEINPUT(scriptinputs, pattern)
        if input ≢ null then
            let Cert_RP := π_2(π_4(input))
            if checksig(Cert.Content, Cert.Sig, s'.PubKey) ≡ null then
                let stop ⟨⟩
            end if
```

```
      let s′.Parameters[Cert] := Cert_RP
      let t := s′.Parameters[t]
      let PID_RP := Multiply(Cert_RP.ID_RP, t)
      let s′.Parameters[PID_RP] := PID_RP
      let Url := ⟨URL, S, IdPDomain, /reqToken, ⟨⟨PID_RP, PID_RP⟩⟩⟩
      let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s′.refXHR⟩
      let s′.phase := expectLoginState
    end if
  case expectReqToken:
    let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      if π_2(input) ≡ Unanthenticated then
        let user ∈ ids
        let Url := ⟨URL, S, IdPDomain, /authentication, ⟨⟩⟩
        let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨username, username⟩, ⟨password, password⟩⟩, s′.
        let s′.phase := expectLoginResult
      end if
      let IDToken := π_2(input)[IDToken]
      let RPOringin := ⟨s′.Parameters[Cert].Enpt, S⟩
      let command := ⟨POSTMESSAGE, target, ⟨IDToken, IDToken⟩, RPOrigin⟩
      let s.phase := stop
    end if
  case expectLoginResult:
    let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      if π_2(input) ≢ LoginSuccess then
        let stop ⟨⟩
      end if
      let PID_RP := s′.Parameters[PID_RP]
      let Url := ⟨URL, S, IdPDomain, /authorize, ⟨⟨PID_RP, PID_RP⟩, ⟨Attr, Attr⟩⟩⟩
      let command := ⟨XMLHTTPREQUEST, Url, GET, ⟨⟩, s′.refXHR⟩
      let s′.phase := expectToken
    end if
  case expectToken:
    let pattern := ⟨XMLHTTPREQUEST, Body, s′.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      let IDToken := π_2(input)[IDToken]
      let RPOringin := ⟨s′.Parameters[Cert].Enpt, S⟩
      let command := ⟨POSTMESSAGE, target, ⟨IDToken, IDToken⟩, RPOrigin⟩
      let s.phase := stop
    end if
end switch
let stop ⟨s′, cookies, localStorage, sessionStorage, command⟩
```

4 Algorithm script_rp

```
Input: ⟨tree, docID, scriptstate, scriptinputs, cookies, ids, secret⟩
let s' := scriptstate
let command := ⟨⟩
let IdPWindow := SUBWINDOW(tree, docnonce).winID
let RPDomain := s'.RPDomain
let IdPOringin := ⟨s'.IdPDomian, S⟩
switch s'.phase do
  case start:
    let Url := ⟨URL, S, RPDomain, /loginSSO, ⟨⟩⟩
    let command := ⟨IFRAME, Url, _SELF⟩
    let s'.phase := expectt
  case expectt:
    let pattern := ⟨POSTMESSAGE, target, ∗, ⟨t, ∗⟩⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      let t := π_2(π_4(input))[t]
      let Url := ⟨URL, S, RPDomain, /startNegotiation, ⟨⟩⟩
      let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨t, t⟩⟩, s'.refXHR⟩
      let s'.phase := expectCert
    end if
  case expectCert:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      let Cert_RP := π_2(input)[Cert_RP]
      let command := ⟨POSTMESSAGE, IdPWindow, ⟨⟨Cert, Cert⟩⟩, IdPOringin⟩
      let s'.phase := expectToken
    end if
  case expectToken:
    let pattern := ⟨POSTMESSAGE, target, ∗, ⟨IDToken, ∗⟩⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      let IDToken := π_2(input)[IDToken]
      let Url := ⟨URL, S, RPDomain, /uploadToken, ⟨⟩⟩
      let command := ⟨XMLHTTPREQUEST, Url, POST, ⟨⟨IDToken, IDToken⟩⟩, s'.refXHR⟩
      let s'.phase := expectLoginResult
    end if
  case expectLoginResult:
    let pattern := ⟨XMLHTTPREQUEST, Body, s'.refXHR⟩
    let input := CHOOSEINPUT(scriptinputs, pattern)
    if input ≢ null then
      if π_2(input) ≡ LoginSuccess then
        let Load Homepage
```

end if
end if
end switch
let stop $\langle s', cookies, localStorage, sessionStorage, command\rangle$

2 Proof of Security

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

Definition 1. *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 $r \in \mathtt{RP}$ that is honest, every RP service token of the form $\langle IDToken,\ Acct\rangle$ recorded in $S^j(r)$.*`serviceTokens`*, the following two conditions are satisfied:*

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

(B) If the request corresponding to $\langle IDToken,\ Acct\rangle$ was sent by some $b \in \mathtt{B}$ which is honest in S^j, then b owns the ID_U which satisfies $Acct = [ID_U]S^j(r).ID_{RP}$.

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

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

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

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

We now proceed to to proof that this is a contradiction. Let $I := \mathtt{governor}(i)$. We know that I is an honest IdP. As such, it never leaks its signing key (see Algorithm 1). Therefore, the signed subterm $Content := \langle PID_{RP}, PID_U, s'.Issuer, Validity \rangle$, $Sig := SigSign(Content, s'.SK)$ and $IDToken := \langle Content, Sig \rangle$ had to be created by the IdP I. An (honest) IdP creates signatures only in Line 48-50 of Algorithm 1.

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

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

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

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

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

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

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

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

Proof. The proof is same as proof of Lemma 8 in SPRESSO. It can be proved that, the $IDToken$ held by the honest *script_idp* is only sent to the origin $\langle Cert_{RP}.Enpt_{RP}, S\rangle$, while the $IDToken.PID_{RP} \equiv [t]Cert_{RP}.ID_{RP}$, and t is the one-time random number. The relation of ID_RP and $Enpt$ is guaranteed by the signature generated by IdP I. The process is shown at Line 9, 16, 19, 21, 38, 39, 59, 60 in Algorithm 3. □

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

Proof. The proof follows the proof of Lemma 9 in SPRESSO. It is proved that *script_rp* of the origin $\langle Cert_{RP}.Enpt_{RP}, S\rangle$ would only sent to the corresponding RP r, which is shown in Algorithm 4. □

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

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

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

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

2.2 Proof of Property B

As stated above, Property B is defined as follows:

Definition 3. *Let $\mathcal{UWS}^{auth}$ be an UPPRESSO web system for authentication analysis. We say that $\mathcal{UWS}^{auth}$ is secure (with respect to Property A) if for every run rho of $\mathcal{UWS}^{auth}$, every state (S^j, E^j, N^j) in rho, every $r \in \texttt{RP}$ that is honest in S^j, every RP service token of the form $\langle IDToken, Acct\rangle$ recorded in $S^j(r)$.`serviceTokens`, with the request corresponding to $\langle IDToken, Acct\rangle$ sent by some $b \in B$ which is honest in S^j, b owns Acct.*

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

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

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

An attacker may allure the honest user to upload the $IDToken \in d_{\emptyset}(S^j(\texttt{attacker}))$ to honest $r \in \texttt{RP}$, so that there may be $Acct \in d_{\emptyset}(S^j(\texttt{attacker}))$. However, while b has already negotiated the PID_{RP} with r, the opener of the *script_idp* must be the *script_rp*. As the t generated at Line 7, Algorithm 3, and PID_{RP} generated at Line 21 in Algorithm 3. The t is only sent to *script_rp* at Line 8 in Algorithm 3, and the *script_rp* receives it at Line 18 in Algorithm 4. The PID_{RP} is sent to the honest IdP at Lines 23 and 50 in Algorithm 3, which is used for generating the $IDToken$.

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

With the above proofs, we now can guarantee that every $\mathcal{UWS}^{auth}$ system satisfies the requirements in Definition 3, therefore $\mathcal{UWS}$ must be secure of Property B.

3 Proof of Privacy

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

3.1 Formal Model of UPPRESSO for Privacy Analysis

Definition 4 (Challenge Browser)**.**

Definition 5 (Deterministic DY Process)**.**

Definition 6 (UPPRESSO Web System for Privacy Analysis). *Let* $\mathcal{UWS} = (\mathcal{W}, \mathcal{S}, script, E^0)$ *be an UPPRESSO web system with* $\mathcal{W} = Hon \cup Web \cup Net$, $Hon = B \cup RP \cup IDP \cup DNS$. *Let* $attacker \in Web$ *be some web attacker. Let* dr *be a domain of* r_1 *or* r_2 *and* $b(dr)$ *be a challenge browser. Let* $Hon\prime := \{b(dr)\} \cup RP \cup DNS$, $Web\prime := Web$, *and* $Net\prime := \emptyset$. *Let* $\mathcal{W}\prime := Hon\prime \cup Web\prime \cup Net\prime$. *We call* $\mathcal{UWS}^{priv}(dr) = (\mathcal{W}\prime, \mathcal{S}\prime, script\prime, E^0, attacker)$ *an UPPRESSO web system for privacy analysis.*

Definition 7 (IdP-Privacy). *Let*

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

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

3.2 Definition of Equivalent Configurations

Let $\mathcal{UWS}_1^{priv}$ and $\mathcal{UWS}_2^{priv}$ be UPPRESSO web system 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) accordingly.

Definition 8 (Challenge Browser).

Definition 9 (Term Equivalence up to Proto-Tags).

Definition 10 (Equivalence of HTTP Requests).

Definition 11 (Extracting Entries from Login Sessions).

Definition 12 (Login Session Token).

Definition 13 (Equivalence of States). *Same as Definition 79 in SPRESSO except that the first condition in Definition 79 in SPRESSO is not applicable.*

Definition 14 (Equivalence of Events). *Same as Definition 80 in SPRESSO except that the forth condition in Definition 80 in SPRESSO is not applicable.*

Definition 15 (Equivalence of Configurations).

3.3 Privacy Proof

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

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

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

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

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

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

Proof. Let $\theta = H = L = \emptyset$.Obviously, both latter conditions are true. For all parties $p \in \mathcal{W}_1 \setminus \{b_1\}$, it is clear that $S_1^0(p) = S_2^0(p)$. Also the states $S_1^0(b_1) = S_2^0(b_2)$ are equal. Therefore, all conditions of Definition 13 are fulfilled. Hence, the initial configurations are α-equivalent. $\square$

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

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

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

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

To induce a processing step, the ci-th message from E_1 or E_2, respectively, is selected.Following Definition 14, 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

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

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

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

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

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

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

There are four possible types of HTTP requests that are accepted by r_1 in Algorithm 2:

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

From the cases in Definition 14, only two can possibly apply here:Case 1a and Case 1e. For both cases, we will now analyze each of the HTTP requests listed above separately.

Definition 14,Case 1a:$e^{(1)}_i \rightleftharpoons e^{(2)}_i$. This case implies $p_2 = r_1 = p_1$. As we see below, for the output events $E^{(1)}_{out}$ and $E^{(2)}_{out}$ (if any) only Case 1a of Definition 14 applies. This implies the nonce of both the incoming HTTP requests and HTTP responses cannot be in H.

- path=/script In this case, the same output event is produced whose message is
$$\langle HTTPResp, n, 200, \langle\rangle, RPScript\rangle \tag{4}$$
We can note that Condition 5 of Definition 14 holds true and, also, (*) applies.The remaining conditions are trivially fulfilled and $E_1\prime$ and $E_2\prime$ are β-equivalent under (θ, H, L).As there are no changes to any state, we have that $S_1\prime$ and $S_2\prime$ are γ-equivalent under (θ, H). No new nonces are chosen, hence $N_1\prime = N_1 = N_2 = N_2\prime$.
- path=/loginSSO In this case, the reason for equivalence holding is similar to the case above since the same output event is produced.
- path=/startNegotiation(derive a PID_rp), Line 9;
- path=/uploadToken(verify ID token, calculate Acct), Line 18.

□