Formal Security Analysis of the OpenID Financial-grade API 2.0

TECHNICAL REPORT – WP 1(A)

Pedram Hosseyni, Ralf Küsters, Tim Würtele
Institute of Information Security – University of Stuttgart, Germany
{pedram.hosseyni, ralf.kuesters, tim.wuertele}@sec.uni-stuttgart.de

CONTENTS

I. Introduction

With the emergence of FinTech companies, interfaces between banks and FinTechs became increasingly important. While early FinTechs were forced to use techniques like screen scraping to deliver their services, pressure from customers and lawmakers, e.g., with the EU's 2019 Payment Services Directive 2 (PSD2), led to widespread adoption of *Open Banking APIs* by banks.

One important open banking standard is the *OpenID Financial-grade API 1.0* (FAPI 1.0). FAPI 1.0 is a profile (i.e., a set of concrete protocol flows with extensions) of the *OAuth 2.0 Authorization Framework* [15], the *OpenID Connect* identity layer [24] and several extensions. FAPI 1.0 was developed and standardized in 2021 by the OpenID Foundation with the support of many large corporations, such as Microsoft and the largest Japanese consulting firm, Nomura Research Institute. The goal was to define a REST/JSON model protected by OAuth with high security guarantees.

The development of FAPI 1.0 was accompanied by a formal security analysis [8] which unveiled some previously unknown attacks. This analysis was conducted using the *Web Infrastructure Model (WIM)*, which has been successfully used to find vulnerabilities in and prove the security of several web applications and standards [4, 10–14].

However, the need for a broader scope and complete interoperability at the interface between client and authorization server as well as interoperable security mechanisms at the interface between client and resource server has led to further, ongoing standardization work on a second version of the FAPI, called *FAPI 2.0*. Where not explicitly stated otherwise, we write FAPI for FAPI 2.0 in the remainder of this report.

Similar to FAPI 1.0., FAPI 2.0 is a profile of the *OAuth 2.0 Authorization Framework* [15], the *OpenID Connect* identity layer [24] and a set of extensions for those, namely *OAuth 2.0 Bearer Tokens* [16], *Proof Key for Code Exchange by OAuth Public Clients (PKCE)* [25], *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens (MTLS)* [2], *OAuth 2.0 Pushed Authorization Requests (PAR)* [20], *OAuth 2.0 Authorization Server Metadata* [17], *OAuth 2.0 Authorization Server Issuer Identification* [26], and *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)* [7].

This report contains the first part of a formal security analysis of FAPI 2.0 in the WIM: The formal model of the FAPI participants, i.e., the user and her browser, authorization server (AS), client, and resource server (RS); based on this formal model, we formulate precise formal security properties for FAPI 2.0. As agreed with the FAPI WG, this report is based on commit 209f58a of the FAPI WG's Bitbucket repository [6].

The remainder of this report is structured as follows: We start with an overview of the FAPI 2.0 (baseline) protocol in Section II. Section III contains some remarks on the specifications which may be of interest for the FAPI WG. This is followed by the formal model of the protocol in Section IV, including the formal models of the FAPI 2.0 clients, authorization servers, and resource servers. We continue with the formal definition of the overall system that we analyze in Section V as well as notes on the attacker model in Section VI. The formal security properties are described in Section VII. The formal model is based on the Web Infrastructure Model, which is included in Appendix A.

II. FAPI 2.0 Overview

FAPI 2.0 is based on the OAuth Authorization Code Grant in which a client, e.g., a FinTech, obtains an access token which allows the client to access a user's resources, e.g., a list of transactions of the user's bank account. Similar to the different profiles in FAPI 1.0, which basically represent different security levels, FAPI 2.0 consists of a *baseline* profile and an *advanced* profile, with the latter adding several mechanisms to achieve non-repudiation.[1] We only consider the baseline profile for this

[1] Note: These profiles have been renamed recently, but as mentioned in the introduction, this report is based on commit 209f58a.

report.

A. Protocol Participants

Being based on OAuth, FAPI defines the same set of protocol participants: *Resource Owners* are entities capable of granting access to a protected resource which is hosted at a *Resource Server*. We usually refer to the resource owner as user. Note, however, that a resource owner is not necessarily a person. *Clients* are applications making requests to protected resources on behalf of a resource owner and with their authorization. *Authorization Servers* are responsible for authenticating resource owners and obtaining their consent to grant clients access to protected resources by issuing access tokens. In addition, authorization servers may – given the user's consent – provide clients with OpenID Connect [24] ID Tokens which contain information about the user.

B. FAPI 2.0 Baseline

Figure 1 depicts a simplified overview of the FAPI 2.0 baseline protocol flow in which the client and AS use DPoP for token binding and the client chooses not to request an ID Token.[2] Note that we treat and model the user and her behavior as part of the browser, and we often use the terms synonymously. An OAuth flow is usually initiated by a user interacting with the client (Step $\boxed{1}$), e.g., by selecting which bank the user wants to connect to the client. However, this step is out of scope of the FAPI 2.0 specification. Conducting an OAuth flow requires knowledge of certain authorization server properties, e.g., URLs of relevant endpoints. The authorization server is required to publish a so called *Authorization Server Metadata Document* at a specific, static path so clients only need to know the domain of the authorization server to access this metadata document (Steps $\boxed{2}$ and $\boxed{3}$). After obtaining all necessary data, the client assembles an *Authorization Request*. This authorization request includes the client's identity at the selected authorization server, a scope value describing the kind of access the client requests, a PKCE challenge to bind the authorization code to the requesting client instance, a redirect URI to which the user will be redirected after authentication at the authorization server and several other parameters. Instead of adding the authorization request parameters to a URL and redirecting the user to that URL – as would be the case with plain OAuth 2.0 –, the client then sends the authorization request directly to the authentication server as a *Pushed Authorization Request (PAR)* (Step $\boxed{4}$). Crucially, this enables the authorization server to authenticate the client and bind the authorization request to this client. In response to the PAR, the authorization server issues a (random) reference, the *request URI* (Step $\boxed{6}$), which is subsequently used by the client to refer to this authorization request. After receiving the request URI, the client instructs the browser to navigate to the authorization server's authorization endpoint and passes as parameters its client identity as well as the request URI (Steps $\boxed{7}$ and $\boxed{8}$).

The user now authenticates herself to the authorization server and confirms to grant the client the requested access to the user's resources (Step $\boxed{9}$), note that this process is out of scope of FAPI. Assuming that the user confirmed the requested access, the authorization server redirects the user back to the client, passing an *authorization code* as well as an issuer value to prevent mix-up attacks [13] (Steps $\boxed{10}$ and $\boxed{11}$). The client now checks whether the issuer value matches the authorization server it intended to use before continuing (Step $\boxed{12}$).

With the authorization code, the client can now obtain an access token by sending a token request with the authorization code to the authorization server's token endpoint (Step $\boxed{13}$). FAPI mandates several security mechanisms at the token endpoint (Step $\boxed{14}$): 1) Clients have to authenticate themselves to the authorization server which has to verify that the client requesting the access token is the same client for which the authorization code has been issued. 2) As part of the PAR (Step $\boxed{4}$), the client sent a PKCE challenge, i.e., a hash $h(c)$ of a random value c. In the token request, the client now has to provide the corresponding PKCE verifier, i.e., the random value c itself. 3) All issued access tokens must be bound to a key pair – either using DPoP as shown in Figure 1 or with mTLS.

If all checks are successful, the authorization server issues an access token to the client (Step $\boxed{15}$). In case the client also needs to know the user's identity (at the authorization server), it can request an ID Token – which will be issued together with the access token – by adding `openid` to the requested scopes in step $\boxed{4}$.

The client can now use the access token to request protected resources from a resource server (Step $\boxed{16}$). Note that the resource server has to verify the aforementioned binding of the access token a key pair to prevent third parties to use a leaked access token (Step $\boxed{17}$). There are a number of ways for the resource server to verify access tokens, most notably *OAuth 2.0 Token Introspection* [23] in which the resource server queries the issuing authorization server to determine the status of an access token. In addition, the resource server also has to verify the DPoP (or mTLS) sender-constraining of the access token. To do so, the resource server needs to learn which public key the client used when receiving the access token (i.e., the key used for DPoP or mTLS). This information can, for example, be conveyed as part of the access token itself (structured token) or via token introspection.

[2] All protocol flow charts are generated with Annex.

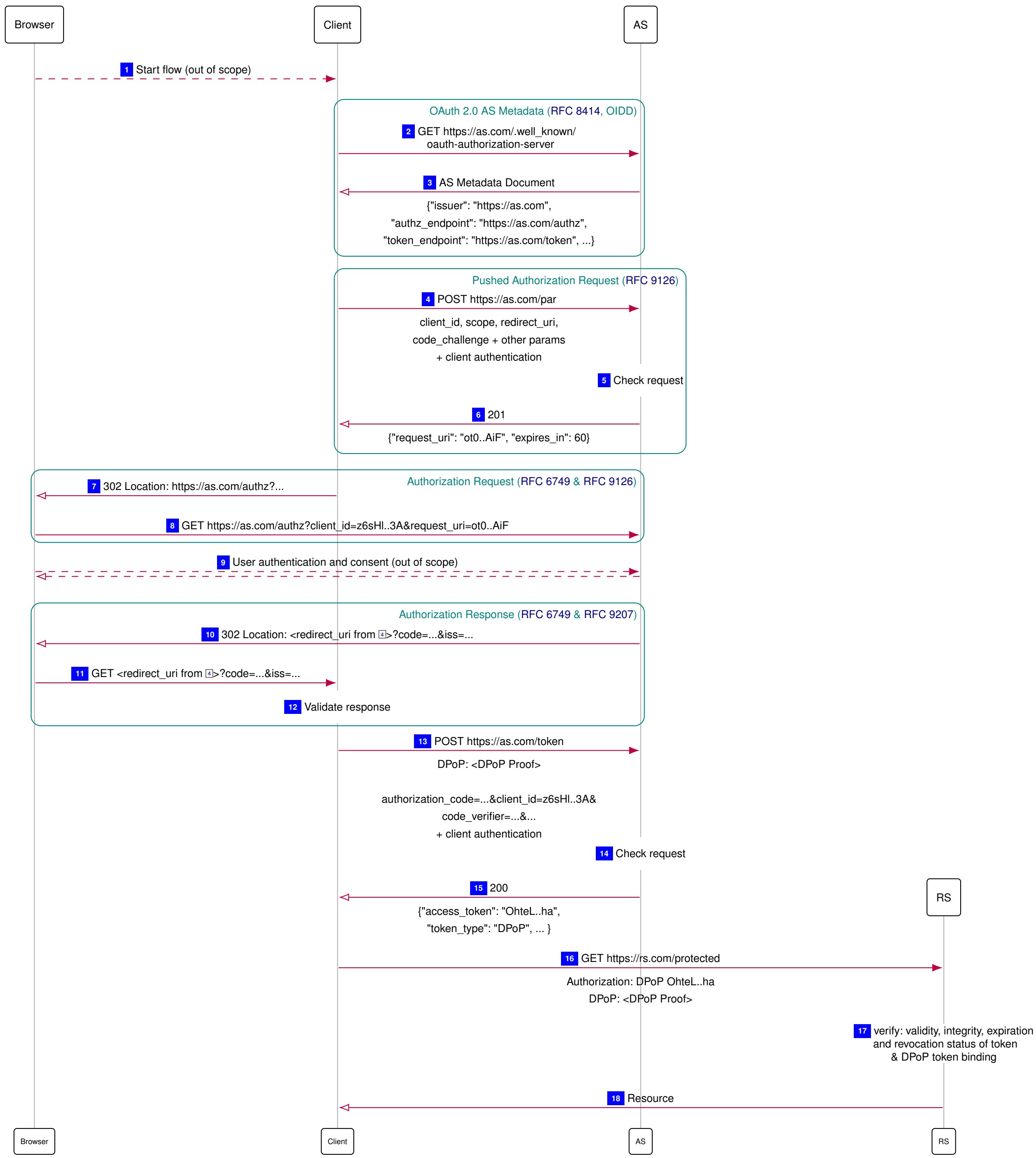

Figure 1. FAPI 2.0 Baseline Flow with DPoP token binding

III. Remarks on the Specifications

This section contains some remarks on the FAPI and related specifications which may be of interest for the OIDF's FAPI WG.

DPoP Authorization Code Binding. Section 10 of the DPoP draft [7] defines an optional `dpop_jkt` authorization parameter. Section 10.1, however, can be read as if an AS supporting PAR [20] and DPoP is required to also support the `dpop_jkt` authorization parameter. As FAPI 2.0 ASs will, in some cases, support both PAR and DPoP, it may be helpful to clarify this.

Security BCP. Section "Differences to FAPI 1.0" of FAPI 2.0 says that FAPI 2.0 implementation "shall adhere to Security BCP", which presumably refers to the "OAuth 2.0 Security Best Current Practice" draft [19]. While some parts of the BCP draft are referenced by FAPI 2.0, it remains unclear whether implementations are required to conform to the full BCP or just the parts explicitly referenced by FAPI 2.0.

Attacker with access to RS requests breaks authorization. An attacker of type A7 or A8, as defined in the FAPI 2.0 attacker model [5], may be able to break the authorization property by simply replaying a resource request if the access token is bound with DPoP.

Tampering with RS responses breaks session integrity. An attacker of type A8 as defined in the FAPI 2.0 attacker model [5] is able to break the second part of the session integrity property by replacing the user's resources with its own, thereby forcing the user to use resources of the attacker.

Preventing Cuckoo's Token Attack. The formal security analysis of FAPI 1.0 [9] uncovered, among others, the Cuckoo's Token Attack. In this attack, the attacker uses a leaked access token which was issued by an honest authorization server for an honest user and injects it into an attacker-initiated protocol flow with an honest client to get access to the honest user's resources. We refer the reader to Section IV-A of [9] for details. FAPI 2.0 aims to achieve authorization (Section 2.1 of [5]) and, as such, should protect against the Cuckoo's Token Attack. In FAPI 2.0, an access token may leak, e.g., to an attacker of type A7 (Section 3.2 of [5]). Similarly, FAPI 2.0 should protect against the access token injection attack described in [9].

While this report does not aim to provide final security results, these attacks might still be possible with FAPI 2.0.

Preventing Authorization Request Leak Attack. Another attack vector on FAPI 1.0 found in the previous formal security analysis is based on leakage of authorization requests and responses (attacker types A3a and A3b in [5]). In this kind of attack, the attacker intercepts the authorization request, logs in under his own identity to receive an authorization code, and redirects the honest user who started the flow, e.g., via a CSRF attack, back to the honest client with that authorization code. This results in the user accessing the attacker's resources, thus breaking session integrity.

Again, while this report does not aim to provide final security results, these attacks might still be possible with FAPI 2.0.

IV. FAPI 2.0 Model

In this section, we provide the full formal model of the FAPI 2.0 participants. We start with the definition of keys and secrets, as well as protocol participants and their identities within the model, followed by how we initialize AS-client relationships and details on how *OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens* [2] is modeled. We continue with the formal models of the FAPI 2.0 clients (Section IV-F), the FAPI 2.0 authorization servers (Section IV-G), and the FAPI 2.0 resource servers (Section IV-H).

A. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into disjoint sets, an infinite set N, and finite sets K_{TLS}, K_{sign}, $\mathsf{Passwords}$, $\mathsf{RScredentials}$, and $\mathsf{Resources}$:

$$\mathcal{N} = N \uplus K_{\text{TLS}} \uplus K_{\text{sign}} \uplus \mathsf{Passwords} \uplus \mathsf{RScredentials} \uplus \mathsf{Resources}$$

These sets are used as follows:

- The set N contains the nonces that are available for the DY processes
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey}\colon \mathsf{Doms} \to K_{\text{TLS}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $tlskeys^p = \langle\{\langle d, \mathsf{tlskey}(d)\rangle \mid d \in \mathsf{dom}(p)\}\rangle$ (i.e., a sequence of pairs).
- The set K_{sign} contains the keys that will be used by ASs for signing id and access tokens and by clients to sign JWTs as well as DPoP proofs. Let $\mathsf{signkey}\colon \mathsf{AS} \cup \mathsf{C} \to K_{\text{sign}}$ be an injective mapping that assigns a (different) signing key to every AS and client.
- The set $\mathsf{Passwords}$ is the set of passwords (secrets) the browsers share with servers. These are the passwords the users use to log in. Let $\mathsf{secretOfID}\colon \mathsf{ID} \to \mathsf{Passwords}$ be a bijective mapping that assigns a password to each identity.

- The set $\mathsf{RScredentials}$ is a set of secrets shared between authorization and resource servers. Resource servers use these to authenticate at authorization servers' token introspection endpoints. Let $\mathsf{secretOfRS}\colon \mathsf{Doms} \times \mathsf{Doms} \rightharpoonup \mathsf{RScredentials}$ be a partial mapping, assigning a secret to some of the resource server – authentication server pairs (with the function arguments in that order).
- The set $\mathsf{Resources}$ models the set of all resources of all resource owners, which we define in the following: For each resource owner, the resource servers shall manage different resources, i.e., no two resource servers shall store the same resource nonce of a resource owner. Let $(\mathsf{Resources}_{id,rs})_{id\in\mathsf{ID},rs\in\mathsf{RS}}$ be a mutually disjoint family of sets, with $\mathsf{Resources}_{id,rs}$ being the set of resources of the resource owner $id \in \mathsf{ID}$ managed by the resource server $rs \in \mathsf{RS}$.
  We define the set of resources of a resource owner $id \in \mathsf{ID}$ as follows:
  $$\mathsf{Resources}_{id} = \bigcup_{rs\in\mathsf{RS}} \mathsf{Resources}_{id,rs}$$
  Finally, we define $\mathsf{Resources}$ as follows:
  $$\mathsf{Resources} = \bigcup_{id\in\mathsf{ID}} \mathsf{Resources}_{id}$$
  Let $\mathsf{resourceOfID}\colon \mathsf{RS} \times \mathsf{ID} \to \mathcal{T}_{\mathcal{N}}$ be a mapping from a resource server rs and identity id to the sequence of nonces $\langle \mathsf{Resources}_{id,rs} \rangle$.

B. Identities and Passwords

Identities consist, similar to email addresses, of a user name and a domain part. For our model, this is defined as follows:

Definition 1. An *identity* i is a term of the form $\langle name, domain\rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$. Let ID be the finite set of identities. We say that an id is *governed* by the DY process to which the domain of the id belongs. This is formally captured by the mappings $\mathsf{governor}\colon \mathsf{ID} \to \mathcal{W}$, $\langle name, domain\rangle \mapsto \mathsf{dom}^{-1}(domain)$ and $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

In addition, we define the mapping $\mathsf{ownerOfSecret}\colon \mathsf{Passwords} \to \mathsf{B}$ that assigns to each password a browser which *owns* this password. Similarly, we define $\mathsf{ownerOfID}\colon \mathsf{ID} \to \mathsf{B}$ as $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (i.e., this identity belongs to the browser).

C. Protocol Participants

We define the following sets of atomic Dolev-Yao processes: AS is the set of processes representing authorization servers. Their relation is described in Section IV-G. RS is the set of processes representing resource servers, described in Section IV-H. C is the set of processes representing clients, described in Section IV-F. Finally, B is the set of processes representing browsers, including their users. They are described in Appendix A-G.

D. Client Registration

Authorization servers and clients have to establish some relationship with each other before starting a FAPI flow. Such a relationship can be established out of band, e.g., via manual configuration. While FAPI also supports the use of dynamic client registration, our model assumes an out of band registration, captured by the following definitions.

Definition 2. A *client information dictionary* is a dictionary of the form $[\mathtt{client\_id}{:}\ clientId, \mathtt{client\_type}{:}\ clientType, \mathtt{mtls\_skey}{:}\ mtlsSkey, \mathtt{jwt\_skey}{:}\ jwtSkey]$ where $clientId \in \mathcal{T}_{\mathcal{N}}$, $clientType \in \{\mathtt{mTLS\_mTLS}, \mathtt{mTLS\_DPoP}, \mathtt{pkjwt\_mTLS}, \mathtt{pkjwt\_DPoP}\}$, $mtlsSkey \in K_{\text{tls}} \cup \{\Diamond\}$, and $jwtSkey \in K_{\text{sign}} \cup \{\Diamond\}$. We further require $jwtSkey \not\equiv \Diamond$ if and only if $clientType \in \{\mathtt{pkjwt\_mTLS}, \mathtt{pkjwt\_DPoP}, \mathtt{mTLS\_DPoP}\}$, as well as $tlsSkey \not\equiv \Diamond$ if and only if $clientType \in \{\mathtt{mTLS\_mTLS}, \mathtt{pkjwt\_mTLS}, \mathtt{mTLS\_DPoP}\}$. Let $\mathsf{ClientInfos}$ be the set of all client information dictionaries.

Definition 3. Let $\mathsf{clientInfo}\colon \mathsf{Doms} \times \mathsf{Doms} \rightharpoonup \mathsf{ClientInfos}$ be a (partial) mapping from authorization server and client domains to client information dictionaries, assigning a client information dictionary to some of the possible authorization server–client pairs with the following restrictions: 1) There are no two clients with the same *clientId* at the same authorization server, formalized as $\forall\, \mathsf{as} \in \mathsf{AS}, d_{as} \in \mathsf{dom}(\mathsf{as}), d_{client}, d'_{client} \in \mathsf{Doms}\colon \mathsf{clientInfo}(d_{as}, d_{client}).\mathtt{client\_id} \equiv \mathsf{clientInfo}(d_{as}, d'_{client}).\mathtt{client\_id} \Rightarrow d_{client} \equiv d'_{client}$, 2) TLS keys are assigned according to tlskey, formalized as $\mathsf{clientInfo}(d_{as}, d_{client}).\mathtt{tlsSkey} \in \{\Diamond, \mathsf{tlskey}(d_{client})\}$, and 3) signing keys are assigned according to $\mathsf{signkey}$, formally expressed as $\mathsf{clientInfo}(d_{as}, d_{client}).\mathtt{jwtSkey} \in \{\Diamond, \mathsf{signkey}(\mathsf{dom}^{-1}(d_{client}))\}$. Note that while this definition requires the client to use the same key to sign JWTs and DPoP proofs for all authorization servers, it allows the client to use a different client type for each authorization server. mTLS keys are different for each of the client's domains.

Definition 4. Let $\mathsf{clientInfoAS}\colon \mathsf{AS} \rightharpoonup \mathcal{T}_{\mathcal{N}}$ be a (partial) mapping from an authorization server to a dictionary. The keys of this dictionary are client IDs and the values *AS client information* dictionaries. We define $\mathsf{clientInfoAS}$ by

$\mathsf{as} \mapsto \langle\{\langle cli.\mathtt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid \exists\ d_{client} \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(\mathsf{as})\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli\}\rangle$ where $\mathsf{as\_cli}(cli) := [\mathtt{client\_type}\colon cli.\mathtt{client\_type}] +^{\langle\rangle}$

$$\begin{cases} [\mathtt{mtls\_key}\colon \mathsf{pub}(cli.\mathtt{mtlsSkey})] & \text{if } cli.\mathtt{client\_type} \equiv \mathtt{mTLS\_mTLS} \\ [\mathtt{mtls\_key}\colon \mathsf{pub}(cli.\mathtt{mtlsSkey}), \mathtt{jwt\_key}\colon \mathsf{pub}(cli.\mathtt{jwtSkey})] & \text{if } cli.\mathtt{client\_type} \in \{\mathtt{pkjwt\_mTLS}, \mathtt{mTLS\_DPoP}\} \\ [\mathtt{jwt\_key}\colon \mathsf{pub}(cli.\mathtt{jwtSkey})] & \text{if } cli.\mathtt{client\_type} \equiv \mathtt{pkjwt\_DPoP} \end{cases}$$

Note: In $\exists\ d_{client} \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(\mathsf{as})\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli$, we refer to values d_{client} and d_{as} for which $\mathsf{clientInfo}(d_{as}, d_{client})$ is defined.

Definition 5. Let $\mathsf{clientInfoClient}\colon \mathsf{C} \rightharpoonup \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ be a (partial) mapping from a client to a dictionary. The keys of this dictionary are AS domains and the values simple dictionaries, containing client type and client ID. clientInfoClient is defined as $c \mapsto \langle\{[d_{as}\colon [\mathtt{client\_id}\colon cli.\mathtt{client\_id}, \mathtt{client\_type}\colon cli.\mathtt{client\_type}]] \mid \exists\ d_{client} \in \mathsf{dom}(c), d_{as} \in \mathsf{Doms}\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli\}\rangle$. Note: In $\exists\ d_{client} \in \mathsf{dom}(c), d_{as} \in \mathsf{Doms}\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli$, we refer to values d_{client} and d_{as} for which $\mathsf{clientInfo}(d_{as}, d_{client})$ is defined.

E. Modeling mTLS

OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens (mTLS) [2] provides a method for both client authentication and token binding. Note that both mechanisms may be used independently of each other.

OAuth 2.0 Mutual TLS Client Authentication makes use of *TLS client authentication*[3] at the token endpoint (in step 13 of Figure 1). In TLS client authentication, not only the server authenticates to the client (as is common for TLS) but the client also authenticates to the server. To this end, the client proves that it knows the private key belonging to a certificate that is either (a) self-signed and pre-configured at the respective AS or that is (b) issued for the respective client id by a predefined certificate authority within a public key infrastructure (PKI).

Token binding means binding an access token to a client such that only this client is able to use the access token at the RS. To achieve this, the AS associates the access token with the certificate used by the client for the TLS connection to the token endpoint. In the TLS connection to the RS (in step 16 of Figure 1), the client then authenticates using the same certificate. The RS accepts the access token only if the client certificate is the one associated with the access token.[4]

The WIM models TLS at a high level of abstraction. An HTTP request is encrypted with the public key of the recipient and contains a symmetric key, which is used for encrypting the HTTP response. Furthermore, the model contains no certificates or public key infrastructures but uses a function that maps domains to their public key.

Figure 2 shows an overview of how we modeled mTLS. The basic idea is that the server sends a nonce encrypted with the public key of the client. The client proves possession of the private key by decrypting this message. In step 1, the client sends its client identifier to the authorization server. The authorization server then looks up the public key associated with the client identifier, chooses a nonce and encrypts it with the public key. As depicted in Step 2, the server additionally includes its public key. When the client decrypts the message, it checks if the public key belongs to the server it wants to send the original message to. This prevents man-in-the-middle attacks, as only the honest client can decrypt the response and as the public key of the server cannot be changed by an attacker. In step 3, the client sends the original request with the decrypted nonce. When the server receives this message, it knows that the nonce was decrypted by the honest client (as only the client knows the corresponding private key) and that the client had chosen to send the nonce to the server (due to the public key included in the response). Therefore, the server can conclude that the message was sent by the honest client.

In effect, this resembles the behavior of the TLS handshake, as the verification of the client certificate in TLS is done by signing all handshake messages [22, Section 7.4.8], which also includes information about the server certificate, which means that the signature cannot be reused for another server. Instead of signing a sequence that contains information about the receiver, in our model, the client checks the sender of the nonce, and only sends the decrypted nonce to the creator of the nonce. In other words, a nonce decrypted by an honest server that gets decrypted by the honest client is never sent to the attacker.

As explained above, the client uses the same certificate it used for the token request when sending the access token to the resource server. While the resource server has to check the possession of corresponding private keys, the validity of the certificate was already checked at the authorization server and can be ignored by the resource server. Therefore, in our model of the FAPI, the client does not send its client id to the resource server, but its public key, and the resource server encrypts the message with this public key.

All messages are sent by the generic HTTPS server model (Appendix A-L), which means that each request is encrypted asymmetrically, and the responses are encrypted symmetrically with a key that was included in the request. For completeness, Figure 3 shows the complete messages, i.e., with the encryption used for transmitting the messages.

[3] As noted in section 7.2 of [2], this extension supports all TLS versions with certificate-based client authentication.

[4] The RS can read this information either directly from the access token if the access token is a signed document, or uses token introspection to retrieve the data from the AS.

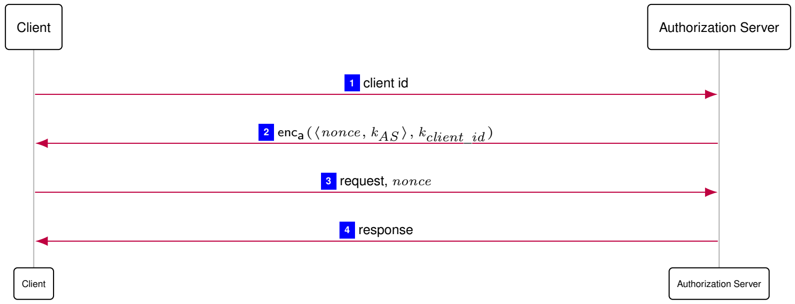

Figure 2. Overview of mTLS

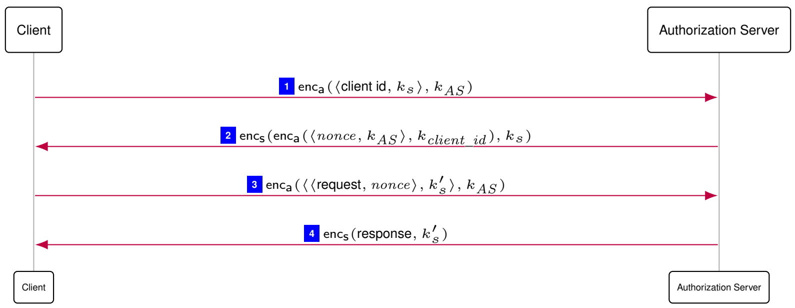

Figure 3. Detailed view on mTLS

F. Clients

A client $c \in \mathsf{C}$ is a web server modeled as an atomic DY process (I^c, Z^c, R^c, s_0^c) with the addresses $I^c := \mathsf{addr}(c)$. Next, we define the set Z^c of states of c and the initial state s_0^c of c.

Definition 6. A *state* $s \in Z^c$ *of a client* c is a term of the form $\langle \mathit{DNSAddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt}, \mathit{keyMapping}, \mathit{tlskeys}, \mathit{sessions}, \mathit{issuerCache}, \mathit{oauthConfigCache}, \mathit{jwksCache}, \mathit{asAccounts}, \mathit{mtlsCache}, \mathit{jwk}\rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 60), $\mathit{sessions} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{issuerCache} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{oauthConfigCache} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{jwksCache} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{asAccounts} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{mtlsCache} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{jwk} \in K_{\text{sign}}$.

An *initial state* s_0^c *of* c is a state of c with $s_0^c.\mathtt{DNSAddress} \equiv I^c$ (see Definition 59), $s_0^c.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^c.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^c.\mathtt{corrupt} \equiv \bot$, $s_0^c.\mathtt{keyMapping}$ being the same as the keymapping for browsers above, $s_0^c.\mathtt{tlskeys} \equiv \mathit{tlskeys}^c$, $s_0^c.\mathtt{sessions} \equiv \langle\rangle$, $s_0^c.\mathtt{issuerCache} \equiv \langle\{\langle \mathit{id}, \pi_2(\mathit{id})\rangle \mid \mathit{id} \in \mathsf{ID}\}\rangle$, $s_0^c.\mathtt{oauthConfigCache} \equiv \langle\rangle$, $s_0^c.\mathtt{jwksCache} \equiv \langle\rangle$, $s_0^c.\mathtt{asAccounts} \equiv \mathsf{clientInfoClient}(c)$ (see Definition 5), $s_0^c.\mathtt{mtlsCache} \equiv \langle\rangle$, and $s_0^c.\mathtt{jwk} \equiv \mathsf{signkey}(c)$.

We now specify the relation R^c: This relation is based on our model of generic HTTPS servers (see Appendix A-L). Hence we only need to specify algorithms that differ from or do not exist in the generic server model. These algorithms are defined in Algorithms 2–6. (Note that in several places throughout these algorithms we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 18).)

The script that is used by the client is described in Algorithm 8. In these scripts, to extract the current URL of a document, the function $\mathsf{GETURL}(\mathit{tree}, \mathit{docnonce})$ is used. We define this function as follows: It searches for the document with the identifier $\mathit{docnonce}$ in the (cleaned) tree tree of the browser's windows and documents. It then returns the URL u of that document. If no document with nonce $\mathit{docnonce}$ is found in the tree tree, $\diamond$ is returned.

Algorithm 1 Relation of a Client R^c – Processing HTTPS Requests

function $\mathsf{PROCESS\_HTTPS\_REQUEST}(m, k, a, f, s')$ $\rightarrow$ **Process an incoming HTTPS request.** Other message types are handled in separate functions. m is the incoming message, k is the encryption key for the response, a is the receiver, f the sender of the message. s' is the current state of the atomic DY process c.
- **if** $m.\mathtt{path} \equiv /$ **then** $\rightarrow$ Serve index page (start flow).
  - **let** $\mathit{headers} := [\mathtt{ReferrerPolicy}{:}\ \mathtt{origin}]$ $\rightarrow$ Set the Referrer Policy for the index page of the Client.
  - **let** $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \mathit{headers}, \langle \mathtt{script\_client\_index}, \langle\rangle\rangle\rangle, k)$ $\rightarrow$ Reply with *script_client_index*.
  - **stop** $\langle\langle f, a, m'\rangle\rangle$, s'
- **else if** $m.\mathtt{path} \equiv /\mathtt{startLogin} \wedge m.\mathtt{method} \equiv \mathtt{POST}$ **then** $\rightarrow$ **Serve the request to start a new login.**
  - **if** $m.\mathtt{headers}[\mathtt{Origin}] \not\equiv \langle m.\mathtt{host}, \mathtt{S}\rangle$ **then**
    - **stop** $\rightarrow$ Check the Origin header for CSRF protection.
  - **let** $\mathit{id} := m.\mathtt{body}$
  - **let** $\mathit{sessionId} := \nu_1$ $\rightarrow$ Session id is a freshly chosen nonce.
  - **let** $s'.\mathtt{sessions}[\mathit{sessionId}] := [\mathtt{startRequest}{:}\ [\mathtt{message}{:}\ m, \mathtt{key}{:}\ k, \mathtt{receiver}{:}\ a, \mathtt{sender}{:}\ f], \mathtt{identity}: \mathit{id}]$
  - **call** $\mathsf{PREPARE\_AND\_SEND\_PAR}(\mathit{sessionId}, a, s')$ $\rightarrow$ Starts or proceed with an authorization flow with the AS. See Algorithm 6.
- **else if** $m.\mathtt{path} \equiv /\mathtt{redirect\_ep}$ **then** $\rightarrow$ **User is being redirected after authentication to the IdP.**
  - **let** $\mathit{sessionId} := m.\mathtt{headers}[\mathtt{Cookie}][\mathtt{sessionId}]$
  - **if** $\mathit{sessionId} \notin s'.\mathtt{sessions}$ **then**
    - **stop**
  - **let** $\mathit{session} := s'.\mathtt{sessions}[\mathit{sessionId}]$ $\rightarrow$ Retrieve session data.
  - **let** $\mathit{id} := \mathit{session}[\mathtt{id}]$
  - **let** $\mathit{issuer} := s'.\mathtt{issuerCache}[\mathit{identity}]$ $\rightarrow$ Issuer cache contains mappings from identites to issuers.
  - **if** $m.\mathtt{parameters}[\mathtt{iss}] \not\equiv \mathit{issuer}$ **then**
    - **stop** $\rightarrow$ Check issuer parameter.
  - **let** $\mathit{oauthConfig} := s'.\mathtt{oauthConfigCache}[\mathit{issuer}]$ $\rightarrow$ Retrieve configuration for issuer.
  - **let** $\mathit{responseType} := \mathit{session}[\mathtt{response\_type}]$ $\rightarrow$ Determines the flow to use, e.g., `code id_token token` for a hybrid flow.
  - **let** $\mathit{data} := m.\mathtt{parameters}$ $\rightarrow$ Authorization code mode: Take data from URL parameters. (The client uses only the authorization code mode.)
  - **let** $s'.\mathtt{sessions}[\mathit{sessionId}][\mathtt{redirectEpRequest}] :=$ $\hookrightarrow$ $[\mathtt{message}{:}\ m, \mathtt{key}{:}\ k, \mathtt{receiver}{:}\ a, \mathtt{sender}{:}\ f]$ $\rightarrow$ Store incoming request for later use in $\mathsf{CHECK\_ID\_TOKEN}$ (Algorithm 5).
  - **call** $\mathsf{SEND\_TOKEN\_REQUEST}(\mathit{sessionId}, m.\mathtt{body}[\mathtt{code}], a, s')$ $\rightarrow$ Retrieve a token from the token endpoint of the IdP.
- **stop**

Algorithm 2 Relation of a Client R^c – Processing HTTPS Responses

function PROCESS_HTTPS_RESPONSE(m, $reference$, $request$, key, a, f, s')
  let $sessionId$:= $reference$[session]
  let $session$:= s'.sessions[$sessionId$]
  let id := $session$[identity]
  let $issuer$:= s'.issuerCache[id]
  if $reference$[responseTo] $\equiv$ CONFIG **then**
    let $oidcc$:= m.body
    if $oidcc$[issuer] $\not\equiv$ $issuer$ **then**
      stop
    let s'.oauthConfigCache[$issuer$] := $oidcc$
    call PREPARE_AND_SEND_PAR($reference$[session], a, s')
  else if $reference$[responseTo] $\equiv$ JWKS **then**
    let s'.jwksCache[$issuer$] := m.body
    call PREPARE_AND_SEND_PAR($reference$[session], a, s')
  else if $reference$[responseTo] $\equiv$ MTLS **then**
    let m_{dec}, k' **such that** $m_{\text{dec}} \equiv \mathsf{dec_a}(m.\text{body}, k') \wedge \langle dom, k' \rangle \in s'.\texttt{tlskeys}$ **if possible; otherwise stop**
    let $mtlsNonce, serverPubKey$ **such that** $m_{\text{dec}} \equiv \langle mtlsNonce, serverPubKey \rangle$ **if possible; otherwise stop**
    if $serverPubKey \equiv s'$.keyMapping[$request$.host] **then** → Verify sender of mTLS nonce
      let $clientId$:= $reference$[client_id]
      let s'.mtlsCache := s'.mtlsCache $+^{\langle\rangle}$ $\langle request.\text{host}, clientId, mtlsNonce \rangle$
  else if $reference$[responseTo] $\equiv$ PAR **then**
    let $requestUri$:= m.body[request_uri]
    let $clientId$:= $session$[client_id]
    let $request$:= s'.sessions[$sessionId$][startRequest]
    let $authEndpoint$:= $oauthConfigCache$[issuer][auth_ep]
    let $authEndpoint$.parameters := [client_id: $clientId$, request_uri: $requestUri$]
    let $headers$:= [Location: $authEndpoint$, ReferrerPolicy: origin]
    let $headers$[Set-Cookie] := [⟨__Host, sessionId⟩: $\langle sessionId, \top, \top, \top \rangle$]
    let $response$:= $\mathsf{enc_s}$(⟨HTTPResp, $request$[message].nonce, 303, $headers$, ⟨⟩⟩, $request$[key])
    let $leak$:= ⟨LEAK, $authEndpoint$⟩ → We assume that the authorization request leaks to the attacker, see [5] and Section VI
    let $leakAddress$ ← IPs
    stop ⟨⟨$request$[sender], $request$[receiver], $response$⟩, ⟨$leakAddress$, $request$[receiver], $leak$⟩⟩, s'
  else if $reference$[responseTo] $\equiv$ TOKEN **then**
  → Non-deterministically decide whether to use the AT or the ID token if the client requested an ID token
    if $session$[scope] $\equiv$ openid **then**
      let $useAccessTokenNow \leftarrow \{\top, \bot\}$
    else
      let $useAccessTokenNow := \top$
    if $useAccessTokenNow \equiv \top$ **then**
      call USE_ACCESS_TOKEN($reference$[session], m.body[access_token], a, s')
    call CHECK_ID_TOKEN($reference$[session], m.body[id_token], s')
  else if $reference$[responseTo] $\equiv$ RESOURCE_USAGE **then**
    let $resource$:= m.body[resource]
    let s'.sessions[$sessionId$][resource] := $resource$
    let $request$:= $session$[redirectEpRequest] → Retrieve stored meta data of the request from the browser to the redir. endpoint
    let $headers$:= [ReferrerPolicy:origin]
    let m' := $\mathsf{enc_s}$(⟨HTTPResp, $request$[message].nonce, 200, $headers$, $resource$⟩, $request$[key])
    stop ⟨⟨$request$[sender], $request$[receiver], m'⟩⟩, s'
  stop

Algorithm 3 Relation of a Client R^c – Request to token endpoint.

```
function SEND_TOKEN_REQUEST(sessionId, code, a, s')
    let session := s'.sessions[sessionId]
    let pkceVerifier := session[code_verifier]
    let identity := session[identity]
    let issuer := s'.issuerCache[identity]
    let credentials := s'.asAccounts[issuer]
    let headers := []
    let body := [grant_type: authorization_code, code: code, redirect_uri: session[redirect_uri]]
    let body[code_verifier] := pkceVerifier    → add PKCE Code Verifier (RFC 7636, Section 4.5)
    let clientId := credentials[client_id]
    let clientSecret := credentials[client_secret]
    let clientType := s'.asAccounts[issuer][client_type]
    let oauthConfig := s'.oauthConfigCache[issuer]
    → We assume that the token endpoint might be misconfigured, see also the A5 Attacker in [5] and Section VI
    let misconfiguredTEp ← {⊤, ⊥}
    if misconfiguredTEp ≡ ⊥ then
        let tokenEndpoint := oauthConfig[token_ep]
    else    → Choose wrong token endpoint.
        let host ← Doms
        let path ← 𝕊
        let parameters ← [𝕊 × 𝕊]
        let tokenEndpoint := ⟨URL, S, host, path, parameters, ⊥⟩
    → Client Authentication:
    if clientType ∈ {mTLS_mTLS, mTLS_DPoP} then    → mTLS client authentication
        let body[client_id] := clientId    → RFC 8705 mandates client_id when using mTLS authentication
        let mtlsNonce such that ⟨tokenEndpoint.host, clientId, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
        let authData := [TLS_AuthN: mtlsNonce]
        let s'.mtlsCache := s'.mtlsCache −⟨⟩ ⟨tokenEndpoint.host, clientId, mtlsNonce⟩
    else if clientType ∈ {pkjwt_mTLS, pkjwt_DPoP} then    → private_key_jwt client authentication
        let jwt := [iss: clientId, sub: clientId, aud: tokenEndpoint]
        let jws := sig(jwt, s'.jwk)
        let authData := [client_assertion: jws]
    → Sender Constraining:
    if clientType ≡ mTLS_mTLS then    → mTLS sender constraining (same nonce as for mTLS authN)
        let mtlsNonce := authData[TLS_AuthN]
        let body[TLS_binding] := mtlsNonce
    else if clientType ≡ pkjwt_mTLS then    → mTLS sender constraining (fresh mTLS nonce)
        let mtlsNonce such that ⟨tokenEndpoint.host, clientId, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
        let s'.mtlsCache := s'.mtlsCache −⟨⟩ ⟨tokenEndpoint.host, clientId, mtlsNonce⟩
        let body[TLS_binding] := mtlsNonce
    else    → Sender constraning using DPoP
        let privKey := s'.jwk    → get private key
        let htu := tokenEndpoint
        let htu[parameters] := ⟨⟩    → Section 4.2 of DPoP: without query
        let htu[fragment] := ⊥    → Section 4.2 of DPoP: without fragment
        let dpopJwt := [headers: [jwk: pub(privKey)]]
        let dpopJwt[payload] := [htm: POST, htu: htu]
        let dpopProof := sig(dpopJwt, privKey)
        let headers[DPoP] := dpopProof    → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
    let body := body +⟨⟩ authData
    let url := tokenEndpoint
    let message := ⟨HTTPReq, ν₂, POST, url.domain, url.path, url.parameters, headers, body⟩
    call HTTPS_SIMPLE_SEND([responseTo: TOKEN, session: sessionId], message, a, s')
```

Algorithm 4 Relation of a Client R^c – Using the access token.

function USE_ACCESS_TOKEN($sessionId$, $token$, a, s')
 let $session := s'.\mathtt{sessions}[sessionId]$
 let $identity := session[\mathtt{identity}]$
 let $issuer := s'.\mathtt{issuerCache}[identity]$
 let $rsDomain \leftarrow s'.\mathtt{oauthConfigCache}[issuer][\mathtt{resource\_servers}]$
 let $url := \langle \mathtt{URL}, \mathtt{S}, rsDomain, /\mathtt{resource}, \langle\rangle, \bot \rangle$
 → AT is sender-constrained, add proof
 let $clientType := s'.\mathtt{asAccounts}[issuer][\mathtt{client\_type}]$
 let $body := []$
 if $clientType \in \{\mathtt{mTLS\_mTLS}, \mathtt{pkjwt\_mTLS}\}$ **then** → mTLS sender constraining
  let $mtlsNonce$ **such that** $\langle rsDomain, clientId, mtlsNonce\rangle \in s'.\mathtt{mtlsCache}$ **if possible; otherwise stop**
  let $body[\mathtt{TLS\_binding}] := mtlsNonce$
  let $s'.\mathtt{mtlsCache} := s'.\mathtt{mtlsCache} -^{\langle\rangle} \langle issuer, clientId, mtlsNonce\rangle$
 else if $clientType \in \{\mathtt{mTLS\_DPoP}, \mathtt{pkjwt\_DPoP}\}$ **then** → DPoP
  let $privKey := s'.\mathtt{jwk}$ → get private key
  let $htu := rsUrl$
  let $htu[\mathtt{parameters}] := \langle\rangle$ → Section 4.2 of DPoP: without query
  let $htu[\mathtt{fragment}] := \bot$ → Section 4.2 of DPoP: without fragment
  let $dpopJwt := [\mathtt{headers}\colon [\mathtt{jwk}\colon \mathsf{pub}(privKey)]]$
  let $dpopJwt[\mathtt{payload}] := [\mathtt{htm}\colon \mathtt{POST}, \mathtt{htu}\colon htu]$
  let $dpopProof := \mathsf{sig}(dpopJwt, privKey)$
  let $headers := [\mathtt{Authorization}: \langle \mathtt{DPoP}, token\rangle]$ → See Section 7.1 of DPoP
  let $headers[\mathtt{DPoP}] := dpopProof$ → add DPoP header; the $dpopJwt$ can be extracted with the $\mathsf{extractmsg}()$ function
 let $url.\mathtt{path} \leftarrow \mathbb{S}$
 let $message := \langle \mathtt{HTTPReq}, \nu_3, \mathtt{POST}, url.\mathtt{domain}, url.\mathtt{path}, url.\mathtt{parameters}, headers, body\rangle$
 call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}\colon \mathtt{RESOURCE\_USAGE}, \mathtt{session}\colon sessionId]$, $message$, a, s')

Algorithm 5 Relation of a Client R^c – Check ID Token.

function CHECK_ID_TOKEN($sessionId$, id_token, s') → **Check ID Token validity and create service session.**
 let $session := s'.\mathtt{sessions}[sessionId]$ → Retrieve session data.
 let $identity := session[\mathtt{identity}]$
 let $issuer := s'.\mathtt{issuerCache}[identity]$ → Retrieve issuer.
 let $oauthConfig := s'.\mathtt{oauthConfigCache}[issuer]$ → Retrieve configuration for that issuer.
 let $credentials := s'.\mathtt{asAccounts}[issuer]$ → Retrieve credentials for issuer.
 let $jwks := s'.\mathtt{jwksCache}[issuer]$ → Retrieve signing keys for issuer.
 let $data := \mathsf{extractmsg}(id\_token)$ → Extract contents of signed ID Token.
 if $data[\mathtt{iss}] \not\equiv issuer$ **then**
  stop → Check the issuer.
 if $data[\mathtt{aud}] \not\equiv credentials[\mathtt{client\_id}]$ **then**
  stop → Check the audience against own client id.
 if $\mathsf{checksig}(id\_token, jwks) \not\equiv \top$ **then**
  stop → Check the signature of the ID Token.
 if $nonce \in session \wedge data[\mathtt{nonce}] \not\equiv session[\mathtt{nonce}]$ **then**
  stop → If a nonce was used, check its value.
 let $s'.\mathtt{sessions}[sessionId][\mathtt{loggedInAs}] := \langle issuer, data[\mathtt{sub}]\rangle$ → User is now logged in. Store user identity and issuer.
 let $s'.\mathtt{sessions}[sessionId][\mathtt{serviceSessionId}] := \nu_4$ → Choose a new service session id.
 let $request := session[\mathtt{redirectEpRequest}]$ → Retrieve stored meta data of the request from the browser to the redir. endpoint in order to respond to it now. The request's meta data was stored in PROCESS_HTTPS_REQUEST (Algorithm 1).
 let $headers := [\mathtt{ReferrerPolicy}\colon \mathtt{origin}]$
 let $headers[\mathtt{Set\text{-}Cookie}] := [\mathtt{serviceSessionId}\colon \langle \nu_4, \top, \top, \top\rangle]$ → Create a cookie containing the service session id.
 let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, request[\mathtt{message}].\mathtt{nonce}, 200, headers, \mathtt{ok}\rangle, request[\mathtt{key}])$
 stop $\langle\langle request[\mathtt{sender}], request[\mathtt{receiver}], m'\rangle\rangle$, s'

Algorithm 6 Relation of a Client R^c – Prepare and send pushed authorization request.

function PREPARE_AND_SEND_PAR($sessionId$, a, s')
  let $redirectUris := \{\langle \mathtt{URL}, \mathtt{S}, d, \mathtt{/redirect\_ep}, \langle\rangle, \bot\rangle \mid d \in \mathsf{dom}(c)\}$ → Set of redirect URIs for all domains of c.
  let $session := s'.\mathtt{sessions}[sessionId]$
  let $identity := session[\mathtt{identity}]$
  let $issuer := s'.\mathtt{issuerCache}[identity]$
  if $issuer \notin s'.\mathtt{oauthConfigCache}$ **then**
    let $host := issuer$
    let $path \leftarrow \{\mathtt{/.well\_known/openid\text{-}configuration}, \mathtt{/.well\_known/oauth\text{-}authorization\text{-}server}\}$
    let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{GET}, host, path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
    call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}\colon \mathtt{CONFIG}, \mathtt{session}\colon sessionId]$, $message$, a, s')
  let $oauthConfig := s'.\mathtt{oauthConfigCache}[issuer]$
  if $issuer \notin s'.\mathtt{jwksCache}$ **then**
    let $url := oauthConfig[\mathtt{jwks\_uri}]$
    let $message := \langle \mathtt{HTTPReq}, \nu_5, \mathtt{GET}, url.\mathtt{host}, url.\mathtt{path}, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
    call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}\colon \mathtt{JWKS}, \mathtt{session}\colon sessionId]$, $message$, a, s')
  → Construct pushed authorization request
  let $parEndpoint := oauthConfig[\mathtt{par\_ep}]$
  let $clientId := s'.\mathtt{asAccounts}[issuer][\mathtt{client\_id}]$
  let $clientType := s'.\mathtt{asAccounts}[issuer][\mathtt{client\_type}]$
  if $clientType \in \{\mathtt{mTLS\_mTLS}, \mathtt{mTLS\_DPoP}\}$ **then** → mTLS client authentication
    let $mtlsNonce$ **such that** $\langle issuer, clientId, mtlsNonce\rangle \in s'.\mathtt{mtlsCache}$ **if possible; otherwise stop**
    let $authData := [\mathtt{TLS\_AuthN}\colon mtlsNonce]$
    let $s'.\mathtt{mtlsCache} := s'.\mathtt{mtlsCache} -^{\langle\rangle} \langle issuer, clientId, mtlsNonce\rangle$
  else if $clientType \in \{\mathtt{pkjwt\_mTLS}, \mathtt{pkjwt\_DPoP}\}$ **then** → `private_key_jwt` client authentication
    let $jwt := [\mathtt{iss}\colon clientId, \mathtt{sub}\colon clientId, \mathtt{aud}\colon parEndpoint]$
    let $jws := \mathsf{sig}(jwt, s'.\mathtt{jwk})$
    let $authData := [\mathtt{client\_assertion}\colon jws]$
  let $pkceVerifier := \nu_{\text{pkce}}$ → Fresh random value
  let $pkceChallenge := \mathsf{hash}(pkceVerifier)$
  let $redirectUri \leftarrow redirectUris$
  let $parData := [\mathtt{response\_type}\colon \mathtt{code}, \mathtt{code\_challenge\_method}\colon \mathtt{S256}, \mathtt{client\_id}\colon clientId,$
    $\hookrightarrow$ $\mathtt{redirect\_uri}\colon redirectUri, \mathtt{code\_challenge}\colon pkceChallenge]$
  let $useOidc \leftarrow \{\top, \bot\}$ → Use of OIDC is optional
  if $useOidc \equiv \top$ **then**
    let $parData[\mathtt{scope}] := \mathtt{openid}$
  let $parData := parData +^{\langle\rangle} authData$
  let $s'.\mathtt{sessions}[sessionId] := s'.\mathtt{sessions}[sessionId] +^{\langle\rangle} parData$
  let $s'.\mathtt{sessions}[sessionId][\mathtt{code\_verifier}] := pkceVerifier$ → Store PKCE randomness in state
  let $authzReq := \langle \mathtt{HTTPReq}, \nu_{\text{parNonce}}, \mathtt{POST}, parEndpoint.\mathtt{host}, parEndpoint.\mathtt{path}, \langle\rangle, \langle\rangle, parData\rangle$
  call HTTPS_SIMPLE_SEND($[\mathtt{responseTo}\colon \mathtt{PAR}, \mathtt{session}\colon sessionId]$, $authzReq$, a, s')

Algorithm 7 Relation of a Client R^c – Handle trigger events.

function PROCESS_TRIGGER(a, s')
  let $action \leftarrow \{\texttt{MTLS\_PREPARE\_AS}, \texttt{MTLS\_PREPARE\_RS}, \texttt{MTLS\_PREPARE\_MISCONFIGURED\_TOKEN\_EP}\}$
  switch $action$ **do**
    case MTLS_PREPARE_AS
      let $server \leftarrow \mathsf{Doms}$
      let $asAcc := s'.\texttt{asAccounts}[server]$
      let $clientId := asAcc[\texttt{client\_id}]$
      let $body := [\texttt{client\_id}: clientId]$
      let $message := \langle \texttt{HTTPReq}, \nu_{\text{mtls}}, \texttt{GET}, server, \texttt{/MTLS-prepare}, \langle\rangle, \langle\rangle, body\rangle$
      call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{MTLS}, \texttt{client\_id}: clientId], message, a, s'$)
    case MTLS_PREPARE_RS
      let $server \leftarrow \mathsf{Doms}$
      let $asAcc := s'.\texttt{asAccounts}[server]$
      let $clientId := asAcc[\texttt{client\_id}]$
      let $resourceServer \leftarrow s'.\texttt{oauthConfigCache}[server][\texttt{resource\_servers}]$
      let $body := [\texttt{client\_id}: clientId]$
      let $message := \langle \texttt{HTTPReq}, \nu_{\text{mtls}}, \texttt{GET}, resourceServer, \texttt{/MTLS-prepare}, \langle\rangle, \langle\rangle, body\rangle$
      call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{MTLS}, \texttt{client\_id}: clientId], message, a, s'$)
    case MTLS_PREPARE_MISCONFIGURED_TOKEN_EP
      let $server \leftarrow \mathsf{Doms}$
      let $asAcc := s'.\texttt{asAccounts}[server]$
      let $clientId := asAcc[\texttt{client\_id}]$
      let $host \leftarrow \mathsf{Doms}$ $\rightarrow$ Non-deterministically choose the domain instead of sending to the correct AS
      let $body := [\texttt{client\_id}: clientId]$
      let $message := \langle \texttt{HTTPReq}, \nu_{\text{mtls}}, \texttt{GET}, host, \texttt{/MTLS-prepare}, \langle\rangle, \langle\rangle, body\rangle$
      call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{MTLS}, \texttt{client\_id}: clientId], message, a, s'$)
  stop

Algorithm 8 Relation of $script\_client\_index$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$ $\rightarrow$ **Script that models the index page of a client.** Users can initiate the login flow or follow arbitrary links. The script receives various information about the current browser state, filtered according to the access rules (same origin policy and others) in the browser.

let $switch \leftarrow \{\texttt{auth}, \texttt{link}\}$ $\rightarrow$ Non-deterministically decide whether to start a login flow or to follow some link.
if $switch \equiv \texttt{auth}$ **then** $\rightarrow$ **Start login flow.**
  let $url := \mathsf{GETURL}(tree, docnonce)$ $\rightarrow$ Retrieve own URL.
  let $id \leftarrow ids$ $\rightarrow$ Retrieve one of user's identities.
  let $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/startLogin}, \langle\rangle, \bot\rangle$ $\rightarrow$ Assemble URL.
  let $command := \langle \texttt{FORM}, url', \texttt{POST}, id, \bot\rangle$ $\rightarrow$ Post a form including the identity to the Client.
  stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$ $\rightarrow$ Finish script's run and instruct the browser to follow the command (form post).
else $\rightarrow$ **Follow link.**
  let $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$ $\rightarrow$ Non-deterministically select protocol (HTTP or HTTPS).
  let $host \leftarrow \mathsf{Doms}$ $\rightarrow$ Non-det. select host.
  let $path \leftarrow \mathbb{S}$ $\rightarrow$ Non-det. select path.
  let $fragment \leftarrow \mathbb{S}$ $\rightarrow$ Non-det. select fragment part.
  let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ $\rightarrow$ Non-det. select parameters.
  let $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$ $\rightarrow$ Assemble URL.
  let $command := \langle \texttt{HREF}, url, \bot, \bot\rangle$ $\rightarrow$ Follow link to the selected URL.
  stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$ $\rightarrow$ Finish script's run and instruct the browser to follow the command (follow link).

G. Authorization Servers

An authorization server as $\in$ AS is a web server modeled as an atomic process $(I^{\mathsf{as}}, Z^{\mathsf{as}}, R^{\mathsf{as}}, s_0^{\mathsf{as}})$ with the addresses $I^{\mathsf{as}} := \mathsf{addr}(\mathsf{as})$. Next, we define the set Z^{as} of states of as and the initial state s_0^{as} of as.

Definition 7. A *state* $s \in Z^{\mathsf{as}}$ *of an AS* as is a term of the form $\langle \mathit{DNSAddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt}, \mathit{keyMapping}, \mathit{tlskeys}, \mathit{registrationRequests}, \mathit{clients}, \mathit{records},$ (signing key) $\mathit{jwk}, \mathit{authorizationRequests}, \mathit{mtlsRequests}\rangle$ with: $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (as in Definition 60), $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$, $\mathit{jwk} \in K_{\mathrm{sign}}$, $\mathit{registrationRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{clients} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{records} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{authorizationRequests} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{mtlsRequests} \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, and $\mathit{rsCredentials} \in \mathcal{T}_{\mathcal{N}}$.

An *initial state* s_0^{as} *of* as is a state of as with $s_0^{\mathsf{as}}.\mathtt{DNSaddress} \equiv I^{\mathsf{as}}$ (see Definition 59), $s_0^{\mathsf{as}}.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\mathtt{corrupt} \equiv \bot$, $s_0^{\mathsf{as}}.\mathtt{keyMapping}$ being the same as the keymapping for browsers above, $s_0^{\mathsf{as}}.\mathtt{tlskeys} \equiv \mathit{tlskeys}^{\mathsf{as}}$, $s_0^{\mathsf{as}}.\mathtt{jwk} \equiv \mathsf{signkey}(\mathsf{as})$ (see Section IV-A), $s_0^{\mathsf{as}}.\mathtt{registrationRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\mathtt{clients} \equiv \mathsf{clientInfoAS}(\mathsf{as})$ (see Definition 4), $s_0^{\mathsf{as}}.\mathtt{records} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\mathtt{authorizationRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\mathtt{mtlsRequests} \equiv \langle\rangle$, and $s_0^{\mathsf{as}}.\mathtt{rsCredentials} \equiv \mathit{rsCreds}$ where $\mathit{rsCreds}$ is a sequence and $\forall c\colon c \in \mathit{rsCreds} \Leftrightarrow (\exists d \in \mathsf{dom}(\mathsf{as}), \mathit{rs} \in \mathsf{Doms}\colon c \equiv \mathsf{secretOfRS}(d, \mathit{rs}))$.

We now specify the relation R^{as}: This relation is based on our model of generic HTTPS servers (see Appendix A-L). We specify algorithms that differ from or do not exist in the generic server model in Algorithms 9 to 11. Algorithm 12 shows the script *script_as_form* that is used by ASs.

Algorithm 9 Relation of AS R^{as} – Processing HTTPS Requests

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
  if $m.\mathtt{path} \equiv$ /.well-known/openid-configuration $\vee$
    ↪ $m.\mathtt{path} \equiv$ /.well-known/oauth-authorization-server **then** → We model both OIDD and RFC 8414.
    let $\mathit{metaData} := [\mathtt{issuer}\colon \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \varepsilon, \langle\rangle, \bot\rangle]$
    let $\mathit{metaData}[\mathtt{auth\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \mathtt{/auth}, \langle\rangle, \bot\rangle$
    let $\mathit{metaData}[\mathtt{token\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \mathtt{/token}, \langle\rangle, \bot\rangle$
    let $\mathit{metaData}[\mathtt{par\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \mathtt{/par}, \langle\rangle, \bot\rangle$
    let $\mathit{metaData}[\mathtt{introspec\_ep}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \mathtt{/introspect}, \langle\rangle, \bot\rangle$
    let $\mathit{metaData}[\mathtt{jwks\_uri}] := \langle \mathtt{URL}, \mathtt{S}, m.\mathtt{host}, \mathtt{/jwks}, \langle\rangle, \bot\rangle$
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, \mathit{metaData}\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if $m.\mathtt{path} \equiv \mathtt{/jwks}$ **then**
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\rangle, \mathsf{pub}(s'.\mathtt{jwk})\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if $m.\mathtt{path} \equiv \mathtt{/auth}$ **then** → Authorization endpoint: Reply with login page.
    if $m.\mathtt{method} \equiv \mathtt{GET}$ **then**
      let $\mathit{data} := m.\mathtt{parameters}$
    else if $m.\mathtt{method} \equiv \mathtt{POST}$ **then**
      let $\mathit{data} := m.\mathtt{body}$
    let $\mathit{clientId} := \mathit{data}[\mathtt{client\_id}]$
    let $\mathit{requestUri} := \mathit{data}[\mathtt{request\_uri}]$ → If there is no $\mathtt{request\_uri}$ in m, R^{as} does not define a transition for m.
    let $\mathit{authzRecord} := s'.\mathtt{authorization\_requests}[\mathit{requestUri}]$
    if $\mathit{authzRecord}[\mathtt{client\_id}] \not\equiv \mathit{clientId}$ **then** → Check binding of request URI to client
      stop
    if $\mathit{clientId} \not\in s'.\mathtt{clients}$ **then**
      stop → Unknown client
    let $s'.\mathtt{authorization\_requests}[\mathit{requestUri}][\mathtt{auth2\_reference}] := \nu_5$
    let $m' := \mathsf{enc_s}(\langle \mathtt{HTTPResp}, m.\mathtt{nonce}, 200, \langle\langle \mathtt{ReferrerPolicy}, \mathtt{origin}\rangle\rangle, \langle \mathtt{script\_idp\_form}, [\mathtt{auth2\_reference}\colon \nu_5]\rangle\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if $m.\mathtt{path} \equiv \mathtt{/auth2} \wedge m.\mathtt{method} \equiv \mathtt{POST} \wedge m.\mathtt{headers}[\mathtt{Origin}] \equiv \langle m.\mathtt{host}, \mathtt{S}\rangle$ **then** → Second step of authorization
    let $\mathit{identity} := m.\mathtt{body}[\mathtt{identity}]$
    let $\mathit{password} := m.\mathtt{body}[\mathtt{password}]$
    if $\mathit{identity}.\mathtt{domain} \not\in \mathsf{dom}(\mathsf{as})$ **then**
      stop → This AS does not manage $\mathit{identity}$
    if $\mathit{password} \not\equiv \mathsf{secretOfID}(\mathit{identity})$ **then**
      stop → Invalid user credentials
    let $\mathit{auth2Reference} := m.\mathtt{body}[\mathtt{auth2\_reference}]$
    let $\mathit{requestUri}$ **such that** $s'.\mathtt{authorization\_requests}[\mathit{requestUri}][\mathtt{auth2\_reference}] \equiv \mathit{auth2Reference}$
      ↪ **if possible; otherwise stop**

```
    let authzRecord := s'.authorization_requests[requestUri]
    let authzRecord[subject] := identity
    let authzRecord[issuer] := m.host
    let authzRecord[code] := ν_1    → Generate a fresh, random authorization code
    let s'.records := s'.records +^⟨⟩ authzRecord
    let responseData := [code: authzRecord[code]]
    if authzRecord[state] ≢ ⟨⟩ then
        let responseData[state] := authzRecord[state]
    let redirectUri := authzRecord[redirect_uri]
    let redirectUri.parameters := redirectUri.parameters ∪ responseData
    let redirectUri.parameters[iss] := authzRecord[issuer]
    let m' := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, redirectUri⟩⟩, ⟨⟩⟩, k)
  → We assume that the authorization response leaks to the attacker, see [5] and Section VI
    let leakMessage := ⟨LEAK, redirectURI⟩
    let leakAddress ← IPs
    stop ⟨⟨f, a, m'⟩, ⟨leakAddress, a, leakMessage⟩⟩, s'
else if m.path ≡ /par ∧ m.method ≡ POST then    → Pushed Authorization Request
    if m.body[response_type] ≢ code ∨ m.body[code_challenge_method] ≢ S256 then
        stop
    let authnResult := AUTHENTICATE_CLIENT(m, s')    → Stops in case of errors/failed authentication
    let clientId := authnResult.1
    let s' := authnResult.2
    let mtlsInfo := authnResult.3
    if clientId ≢ m.body[client_id] then
        stop    → Key used in client authentication is not registered for m.body[client_id]
    let redirectUri := m.body[redirect_uri]    → Clients are required to send redirect_uri with each request
    if redirectUri ≡ ⟨⟩ then
        stop
    if redirectUri.protocol ≢ S then
        stop
    let codeChallenge := m.body[code_challenge]    → PKCE challenge
    if codeChallenge ≡ ⟨⟩ then
        stop    → Missing PKCE challenge
    let requestUri := ν_4    → Choose random URI
    let authzRecord := [client_id: clientId]
    let authzRecord[state] := m.body[state]
    let authzRecord[scope] := m.body[scope]
    if nonce ∈ m.body then
        let authzRecord[nonce] := m.body[nonce]
    let authzRecord[redirect_uri] := redirectUri
    let authzRecord[code_challenge] := codeChallenge
    let s'.authorizationRequests[requestUri] := authzRecord    → Store data linked to requestUri
    let m' := enc_s(⟨HTTPResp, m.nonce, 201, ⟨⟩, [request_uri: requestUri]⟩, k)
    stop ⟨⟨f, a, m'⟩⟩, s'
else if m.path ≡ /token ∧ m.method ≡ POST then
    if m.body[grant_type] ≢ authorization_code then
        stop
    let authnResult := AUTHENTICATE_CLIENT(m, s')    → Stops in case of errors/failed authentication
    let clientId := authnResult.1
    let s' := authnResult.2
    let mtlsInfo := authnResult.3
    let code := m.body[code]
    let codeVerifier := m.body[code_verifier]
    if code ≡ ⟨⟩ ∨ codeVerifier ≡ ⟨⟩ then
        stop    → Missing code or code_verifier
    let record, ptr such that record ≡ s'.records.ptr ∧ record[code] ≡ code ∧ code ≢ ⊥ if possible; otherwise stop
    if record[client_id] ≢ clientId then
        stop
    if record[code_challenge] ≢ hash(codeVerifier) ∨ record[redirect_uri] ≢ m.body[redirect_uri] then
        stop    → PKCE verification failed or URI mismatch
```

```
        let clientType := s'.clients[clientId][client_type]
        if clientType ≡ pkjwt_DPoP ∨ clientType ≡ mTLS_DPoP then    → DPoP token binding
            let tokenType := DPoP
            let dpopProof := m.headers[DPoP]
            let dpopJwt := extractmsg(dpopProof)
            let verificationKey := dpopJwt[headers][jwk]
            if checksig(dpopProof, verificationKey) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
                stop    → Invalid DPoP signature (or empty jwk header)
            let dpopClaims := dpopJwt[payload]
            let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
            if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
                stop    → DPoP claims do not match corresponding message
            let cnfContent := [jkt: hash(verificationKey)]
        else if clientType ≡ pkjwt_mTLS ∨ clientType ≡ mTLS_mTLS then    → mTLS token binding
            let tokenType := bearer
            let mtlsNonce := m.body[TLS_binding]
            if clientType ≡ mTLS_mTLS then    → Client used mTLS authentication, reuse data from authentication
                if mtlsNonce ≢ mtlsInfo.1 then
                    stop    → Client tried to use different mTLS key for authentication and token binding
            else    → Client did not use mTLS authentication
                let mtlsInfo such that mtlsInfo ∈ s'.mtlsRequests[clientId] ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
                let s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] −⟨⟩ mtlsInfo
            let mTlsKey := mtlsInfo.2    → mTLS public key of client
            let cnfContent := [x5t#S256: hash(mTlsKey)]
        else
            stop    → Client used neither DPoP nor mTLS
        let s'.records.ptr[code] := ⊥    → Invalidate code
        let atType ← {JWT, opaque}    → The AS chooses randomly whether it issues a structured or an opaque access token
        if atType ≡ JWT then    → Structured access token
            let accessTokenContent := [cnf: cnfContent, sub: record[subject]]
            let accessToken := sig(accessTokenContent, s'.jwk)
        else    → Opaque access token
            let accessToken := ν_2    → Fresh random value
        let s'.records.ptr[access_token] := accessToken    → Store for token introspection
        let s'.records.ptr[cnf] := cnfContent    → Store for token introspection
        let body := [access_token: accessToken, token_type: tokenType]
        if record[scope] ≡ openid then    → Client requested ID token
            let idTokenBody := [iss: ⟨URL, S, record[issuer], ε, ⟨⟩, ⊥⟩]
            let idTokenBody[sub] := record[subject]
            let idTokenBody[aud] := record[client_id]
            if nonce ∈ record then
                let idTokenBody[nonce] := record[nonce]
            let idToken := sig(idTokenBody, s'.jwk)
            let body[id_token] := idToken
        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'
    else if m.path ≡ /introspect ∧ m.method ≡ POST ∧ token ∈ m.body then
        let rsCredentials such that ⟨Basic, rsCredentials⟩ ≡ m.headers[Authorization] if possible; otherwise stop
        if rsCredentials ∉ s'.rsCredentials then
            stop    → Resource server authentication failed
        let token := m.body[token]
        let record such that record ∈ s'.records ∧ record[access_token] ≡ token if possible; otherwise let record := ◇
        if record ≡ ◇ then    → Unknown token
            let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [active: ⊥]⟩, k)
        else    → token was issued by this AS
            let body := [active: ⊤, cnf: record[cnf]]    → cnf claim contains hash of token binding key
            let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'
    else if m.path ≡ /MTLS-prepare then    → See Section IV-E
        let clientId := m.body[client_id]
        let mtlsNonce := ν_3
        let clientKey := s'.clients[clientId][mtls_key]
        let s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] +⟨⟩ ⟨mtlsNonce, clientKey⟩
        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, keyMapping[m.host]⟩, clientKey)⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'
    stop    → Request was malformed or sent to non-existing endpoint.
```

Algorithm 10 Relation of AS R^{as} – Client Authentication

function AUTHENTICATE_CLIENT(m, s') $\quad\rightarrow$ Check client authentication in message m. Stops the current processing step in case of errors or failed authentication.
  if $\texttt{client\_assertion} \in m.\texttt{body}$ **then** $\quad\rightarrow$ `private_key_jwt` client authentication
    let $jwts := m.\texttt{body}[\texttt{client\_assertion}]$
    let $clientId, verificationKey$ **such that** $verificationKey \equiv s'.\texttt{clients}[clientId][\texttt{jwt\_key}] \wedge$
      $\hookrightarrow$ $\mathsf{checksig}(jwts, verificationKey) \equiv \top$ **if possible; otherwise stop**
    let $clientInfo := s'.\texttt{clients}[clientId]$
    let $clientType := clientInfo[\texttt{client\_type}]$
    if $clientType \not\equiv \texttt{pkjwt\_mTLS} \wedge clientType \not\equiv \texttt{pkjwt\_DPoP}$ **then**
      stop $\quad\rightarrow$ Client authentication type mismatch
    let $jwt := \mathsf{extractmsg}(jwts)$
    if $jwt[\texttt{iss}] \not\equiv clientId \vee jwt[\texttt{sub}] \not\equiv clientId$ **then**
      stop
    if $jwt[\texttt{aud}] \not\equiv \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, \texttt{/token}, \langle\rangle, \bot\rangle \wedge jwt[\texttt{aud}] \not\equiv \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, \varepsilon, \langle\rangle, \bot\rangle$
      $\hookrightarrow$ $\wedge\ jwt[\texttt{aud}] \not\equiv \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, \texttt{/par}, \langle\rangle, \bot\rangle$ **then**
      stop $\quad\rightarrow$ `aud` claim value is neither token, nor PAR endpoint nor AS issuer identifier
  else if $\texttt{TLS\_AuthN} \in m.\texttt{body}$ **then** $\quad\rightarrow$ mTLS client authentication
    let $clientId := m.\texttt{body}[\texttt{client\_id}]$ $\quad\rightarrow$ RFC 8705 mandates `client_id` when using mTLS authentication
    let $mtlsNonce := m.\texttt{body}[\texttt{TLS\_AuthN}]$
    let $mtlsInfo$ **such that** $mtlsInfo \in s'.\texttt{mtlsRequests}[clientId] \wedge mtlsInfo.1 \equiv mtlsNonce$ **if possible; otherwise stop**
    let $clientInfo := s'.\texttt{clients}[clientId]$
    let $clientType := clientInfo[\texttt{client\_type}]$
    if $clientType \not\equiv \texttt{mTLS\_mTLS} \wedge clientType \not\equiv \texttt{mTLS\_DPoP}$ **then**
      stop $\quad\rightarrow$ Client authentication type mismatch
    let $s'.\texttt{mtlsRequests}[clientId] := s'.\texttt{mtlsRequests}[clientId] -^{\langle\rangle} mtlsInfo$
  else
    stop $\quad\rightarrow$ Unsupported client (authentication) type
  if $clientType \equiv \texttt{mTLS\_mTLS} \vee clientType \equiv \texttt{mTLS\_DPoP}$ **then**
    return $\langle clientId, s', mtlsInfo\rangle$
  else
    return $\langle clientId, s', \bot\rangle$ $\quad\rightarrow$ `private_key_jwt` client authentication, i.e., no mTLS info

Algorithm 11 Relation of AS R^{as} – Processing other messages.

function PROCESS_OTHER(m, a, f, s')
  stop

Algorithm 12 Relation of $script\_as\_form$: A login page for the user.

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$
let $url := \mathsf{GETURL}(tree, docnonce)$
let $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, \texttt{/auth2}, \langle\rangle, \bot\rangle$
let $formData := scriptstate$
let $identity \leftarrow ids$
let $secret \leftarrow secrets$
let $formData[\texttt{identity}] := identity$
let $formData[\texttt{password}] := secret$
let $command := \langle \texttt{FORM}, url', \texttt{POST}, formData, \bot\rangle$
stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$

H. Resource Servers

A resource server $rs \in \mathsf{RS}$ is a web server modeled as an atomic process $(I^{rs}, Z^{rs}, R^{rs}, s_0^{rs})$ with the addresses $I^{rs} := \mathsf{addr}(rs)$. The set of states Z^{rs} and the initial state s_0^{rs} of rs are defined in the following.

Definition 8. A *state* $s \in Z^{rs}$ *of a resource server* rs is a term of the form $\langle \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt}, \mathit{keyMapping}, \mathit{tlskeys}, \mathit{mtlsRequests}$ (sequence of terms), $\mathit{resourceNonces}$ (dict from ID to sequence of nonces), ids (sequence of ids), asDom (domain), $\mathit{as\_introspect\_ep}$ (URL), $\mathit{authServKey}, \mathit{rsCredentials}\rangle$ with $\mathit{DNSaddress} \in \mathsf{IPs}$, $\mathit{pendingDNS} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{pendingRequests} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{keyMapping} \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{tlskeys} \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (all former components as in Definition 60), $\mathit{mtlsRequests} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{pendingResponses} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{resourceNonces} \in [\mathsf{ID} \times \mathcal{T}_{\mathcal{N}}]$, $\mathit{ids} \subset \mathsf{ID}$, $\mathit{asDom} \in \mathsf{Doms}$, $\mathit{as\_introspect\_ep} \in \mathtt{URL}$, $\mathit{authServKey} \in \mathcal{N}$, and $\mathit{rsCredentials} \in \mathcal{T}_{\mathcal{N}}$.

An *initial state* s_0^{rs} *of* rs is a state of rs with $s_0^{rs}.\mathtt{pendingDNS} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{pendingRequests} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{corrupt} \equiv \bot$, $s_0^{rs}.\mathtt{keyMapping}$ being the same as the keymapping for browsers, $s_0^{rs}.\mathtt{tlskeys} \equiv \mathit{tlskeys}^{rs}$, $s_0^{rs}.\mathtt{mtlsRequests} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{pendingResponses} \equiv \langle\rangle$, $s_0^{rs}.\mathtt{ids} \subset^{\langle\rangle} \langle\mathsf{ID}\rangle$ being the sequence of identities (resource owners) for which the resource server manages resources, $s_0^{rs}.\mathtt{resourceNonces}$ being a dictionary storing the sequence of resource nonces for each identity, i.e., $\forall id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}\colon s_0^{rs}.\mathtt{resourceNonces}[id] := \mathsf{resourceOfID}(rs, id)$, $s_0^{rs}.\mathtt{asDom} \in \mathsf{dom}(as)$ for some $as \in \mathsf{AS}$ being a domain of the authorization server that the resource server supports, $s_0^{rs}.\mathtt{as\_introspect\_ep} \equiv \langle \mathtt{URL}, \mathtt{S}, s_0^{rs}.\mathtt{asDom}, /\mathtt{introspect}, \langle\rangle, \bot\rangle$ being the URL of the introspection endpoint of the authorization server, $s_0^{rs}.\mathtt{authServKey}$ being the verification key for the authorization server $s_0^{rs}.\mathtt{asDom}$, and $s_0^{rs}.\mathtt{rsCredentials}$ being a sequence s.t. $\forall c\colon c \in^{\langle\rangle} s_0^{rs}.\mathtt{rsCredentials} \Leftrightarrow (\exists \mathit{rsDom} \in \mathsf{dom}(rs)\colon c \equiv \mathsf{secretOfRS}(s_0^{rs}.\mathtt{asDom}, \mathit{rsDom}))$, i.e., the secrets used by the resource server for authenticating at the authorization server.

The relation R^{rs} is again based on the generic HTTPS server model (see Appendix A-L), for which the algorithms used for processing HTTP requests and responses are defined in Algorithm 13 and Algorithm 14.

Remarks: To model the protected resource access, we use a set of nonces $\mathit{resourceNonces}$ representing access to a resource. Furthermore, we require that $\forall id \in s_0^{rs}.\mathtt{ids} : \mathsf{governor}(id) \equiv s_0^{rs}.\mathtt{asDom}$, i.e., the resource server contains only resources of identities that are governed by the authorization server $s_0^{rs}.\mathtt{asDom}$.

Algorithm 13 Relation of RS R^{rs} – Processing HTTPS Requests

```
function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
    if m.path ≡ /MTLS-prepare then
        let mtlsNonce := ν_1
        let clientKey := m.body[pub_key]    → Certificate is not required to be checked [2, Section 4.2]
        let s'.mtlsRequests := s'.mtlsRequests +⟨⟩ ⟨mtlsNonce, clientKey⟩
        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, keyMapping(m.host)⟩, clientKey)⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'

    else if m.path ≡ /resource then
        if Authorization ∈ m.headers then
            let authnScheme := m.headers[Authorization].1
            let accessToken := m.headers[Authorization].2
            if authnScheme ≡ bearer then    → mTLS sender constraining
                let mtlsNonce := m.body[TLS_binding]
                let mtlsInfo such that mtlsInfo ∈⟨⟩ s'.mtlsRequests ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
                let s'.mtlsRequests := s'.mtlsRequests −⟨⟩ mtlsInfo
                let mtlsKey := mtlsInfo.2
                let cnfValue := [x5t#S256: hash(mTlsKey)]
            else if authnScheme ≡ DPoP then    → DPoP sender constraining
                let dpopProof := m.headers[DPoP]
                let dpopJwt := extractmsg(dpopProof)
                let verificationKey := dpopJwt[headers][jwk]
                if checksig(dpopProof, verificationKey) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
                    stop    → Invalid DPoP signature (or empty jwk header)
                let dpopClaims := dpopJwt[payload]
                let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
                if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
                    stop    → DPoP claims do not match corresponding message
                let cnfValue := [jkt: hash(verificationKey)]
            else
                stop    → Wrong Authorization header value
            let accessTokenContent such that accessTokenContent ≡ extractmsg(accessToken)
              ↪ if possible; otherwise let accessTokenContent := ◇
            if accessTokenContent ≡ ◇ then    → Not a structured AT, do Token Introspection
        → Store values for the pending request (needed when the resource server gets the introspection response)
                let requestId := ν_2
                let pendingResponses[requestId] := [expectedCNF: cnfValue, requestingClient: f,
                  ↪ originalRequest: m, originalRequestKey: k]
                let url := s'.as_introspect_ep
                let rsCred ← s'.rsCredentials    → Choose a secret for authenticating at the AS (see also sec. 2.1 of RFC 7662)
                let headers := [Authorization: ⟨Basic, rsCred⟩]
                let body := [token: accessToken]
                let message := ⟨HTTPReq, ν_3, POST, url.domain, url.path, url.parameters, headers, body⟩
                call HTTPS_SIMPLE_SEND([responseTo: TOKENINTROSPECTION, requestId: requestId], message, a, s')
            else    → Check structured AT
                if cnfValue.1 ≢ accessTokenContent[cnf].1 ∨ cnfValue.2 ≢ accessTokenContent[cnf].2 then
                    stop    → AT is bound to a different key
                if checksig(accessToken, s'.authServKey) ≢ ⊤ then
                    stop    → Verification of AT signature failed
            let id := accessTokenContent[sub]
            if id ∉⟨⟩ s'.ids then
                stop    → RS does not manage resources of this RO
        → Token binding successfully checked, the RS gives access to a resource of the identity
            let resource ← s'.resourceNonces[id]
            let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [resource:resource]⟩, k)
        → FAPI 2.0 aims to provide security even if the resource request leaks, see [5] and Section VI. In our model, we leak the request
          when the resource server is sending the response.
            let leakedResourceRequest := ⟨LEAK, m⟩
            let leakAddress ← IPs
            stop ⟨⟨f, a, m'⟩, ⟨leakAddress, a, leakedResourceRequest⟩⟩, s'
        else
            stop    → Expected AT in Authorization header
```

Algorithm 14 Relation of a Resource Server R^{rs} – Processing HTTPS Responses

function PROCESS_HTTPS_RESPONSE(m, $reference$, $request$, key, a, f, s')
    if $reference[\texttt{responseTo}] \equiv \texttt{TOKENINTROSPECTION}$ **then**
        let $pendingRequestInfo := s'.\texttt{pendingResponses}[reference[\texttt{requestId}]]$
        let $clientAddress := pendingRequestInfo[\texttt{requestingClient}]$
        let $expectedCNF := pendingRequestInfo[\texttt{expectedCNF}]$
        let $originalRequest := pendingRequestInfo[\texttt{originalRequest}]$
        let $originalRequestKey := pendingRequestInfo[\texttt{originalRequestKey}]$
        if $m.\texttt{body}[\texttt{active}] \not\equiv \top$ **then**
            stop → Access token was invalid
        let $responseCNF := m.\texttt{body}[\texttt{cnf}]$
        if $responseCNF.1 \not\equiv expectedCNF.1 \vee responseCNF.2 \not\equiv expectedCNF.2$ **then**
            stop → Access token was bound to a different key
        let $id := \mathsf{m}.\texttt{body}[\texttt{sub}]$
        if $id \not\in^{\langle\rangle} s'.\texttt{ids}$ **then**
            stop → RS does not manage resources of this RO
        → Token binding successfully checked, the RS gives access to a resource of the identity
        let $resource \leftarrow s'.resourceNonce[id]$
        let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, originalRequest.\texttt{nonce}, 200, \langle\rangle, [\texttt{resource}{:}resource]\rangle, originalRequestKey)$
        → FAPI 2.0 aims to provide security even if the resource request leaks, see [5] and Section VI. In our model, we leak the request when the resource server is sending the response.
        let $leakedResourceRequest := \langle \texttt{LEAK}, originalRequest\rangle$
        let $leakAddress \leftarrow \mathsf{IPs}$
        stop $\langle\langle f, a, m'\rangle, \langle leakAddress, a, leakedResourceRequest\rangle\rangle$, s'

V. FAPI 2.0 Web System

The formal model of the FAPI is a web sytem as defined in Section V.

A web system $\mathcal{FAPI} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is called a *FAPI web system with a network attacker*. The components of the web system are defined in the following.

- $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process (in Net), a finite set B of web browsers, a finite set C of web servers for the clients, a finite set AS of web servers for the authorization servers and a finite set RS of web servers for the resource servers, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$. DNS servers are subsumed by the network attacker and are therefore not modeled explicitly.
- $\mathcal{S}$ contains the scripts shown in Table I, with string representations defined by the mapping script.
- E^0 contains only the trigger events.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
$script\_client\_index$	`script_client_index`
$script\_as\_form$	`script_as_form`

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

For representing access to resources within the formal model, we specify an infinite sequence of nonces N_{resource}. We call these nonces *resource access nonces*.

VI. Attacker Model

The FAPI 2.0 aims to be secure under a strong attacker model as defined in [5]. In the following, we describe how the assumptions on the attacker are incorporated into the FAPI model.

A. A1 and A2 Web and Network Attacker

The goal of the analysis is to prove the security properties in the presence of a network attacker (A2). As network attackers also subsume web attackers, the statements proved about the A2 network attacker would also be true for the A1 web attacker (including the A1a web attacker participating as an authorization server).

B. A3a Attacker - Authorization Request Leakage

The FAPI aims to be secure even if the authorization request leaks to the attacker. We model the leakage of the authorization request in Lines 30-32 of Algorithm 2, where the client sends the request (containing, in particular, the client identifier and the PAR request URI value) to an arbitrary IP address. As this message is sent in plain, it leaks to both the network attacker and the web attacker.

C. A3b Attacker - Authorization Response Leakage

The leakage of the authorization response is modeled by leaking the response at the authorization server. When the authorization server model creates the response, it sends out the value of the location redirect header (which, in particular, contains the authorization code) in plain to an arbitrary address in Lines 50-52 of Algorithm 9.

D. A5 Attacker - Token Endpoint Configuration

The A5 attacker can change the token endpoint that a client is using to an endpoint controlled by the attacker.

In the client model, when sending a token request, the client non-deterministically chooses the token endpoint to be either the correct endpoint obtained from the OAuth Metadata (Line 16 of Algorithm 3) or a non-deterministically chosen URL (Lines 18-21 of Algorithm 3).

E. A7 Attacker - Resource Request and Response Leakage

FAPI 2.0 aims to be secure even if the requests and responses to the resource server leak. We model the leakage of the resource request at the resource server in Line 54 of Algorithm 13 and Line 20 of Algorithm 14.

We note that the leakage of the resource response would directly contradict the Authorization goal stated in [5], i.e., "FAPI 2.0 profiles shall ensure that no attacker can access resources belonging to a user." Thus, the leakage of the resource response is not incorporated into the model.

F. A8 Attacker - Resource Response Tampering

An attacker that can tamper with the resource responses can return the attacker's resources to the user. Thus, this requirement contradicts the Session Integrity for Authorization goal stated in [5], i.e., "FAPI 2.0 profiles shall ensure that no attacker is able to force a user to use resources of the attacker."

VII. Security Properties

The FAPI is an authorization and authentication protocol, thus, both these goals need to be achieved securely. In addition, we define integrity properties. In the following, we give a brief informal description of the security properties, followed by the formal definitions of these properties. We note that the FAPI aims to provide these security properties as specified in the FAPI 2.0 Attacker Model [5].

Informally, the **authorization** property that we formalize below states that the attacker cannot access resources of honest users in an unautorized manner. The attacker, for example, cannot access resources of an honest user by tricking the FAPI client or FAPI authorization server.

The **authentication** property states that the attacker cannot log in at a FAPI client under the account of an honest user.

The two **session integrity** properties state that (1) an honest user, after logging in, is indeed logged in under their own account and not under the account of an attacker, and (2) similarly, that an honest user is accessing their own resources and not the resources of the attacker.

A. Authorization

Intuitively, authorization means that an attacker should not be able to get read or write access to a resource of an honest identity.

More precisely, we require that if an honest resource server provides access to a resource belonging to an honest user whose identity is governed by an honest authorization server, then this access is not provided to the attacker. This includes the case that the resource is not directly accessed by the attacker, but also that no honest client provides the attacker access to such a resource.

Definition 9 (Authorization Property). We say that the FAPI web system with a network attacker $\mathcal{FAPI}$ *is secure w.r.t. authorization* iff for every run ρ of $\mathcal{FAPI}$, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S every identity $id \in \mathsf{ID}^{as}$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S, every client $c \in \mathsf{C}$ that is honest in S with client id $clientId$ issued to c by as, every resource server $rs \in \mathsf{RS}$ that is honest in S such that $id \in s_0^{rs}.\mathtt{ids}$, $s_0^{rs}.\mathtt{authServ} \in \mathsf{dom}(as)$ and with $dom_{rs} \in s_0^{as}.\mathtt{resource\_servers}$ (with $dom_{rs} \in \mathsf{dom}(rs)$), every access token t associated with c, as and id and every resource access nonce $r \in s_0^{rs}.\mathtt{resourceNonce}[id]$ it holds true that:

If r is contained in a response to a request m sent to rs with $t \equiv m.header[\mathtt{Authorization}]$, then r is not derivable from the attackers knowledge in S (i.e., $r \notin d_{\emptyset}(S(\mathsf{attacker}))$).

B. Authentication

Intuitively, an attacker should not be able to log in at an honest client under the identity of an honest user, where the identity is governed by an honest authorization server. All relevant participants are required to be honest, as otherwise, the attacker can trivially log in at a client, for example, if the attacker controls the authorization server that governs the identity.

Definition 10 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some client c* in a configuration (S, E, N) of a run ρ of a FAPI web system iff there exists some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(c).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(c).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n$.

Definition 11 (Authentication Property). We say that the FAPI web system with a network attacker $\mathcal{FAPI}$ *is secure w.r.t. authentication* iff for every run ρ of $\mathcal{FAPI}$, every configuration (S, E, N) in ρ, every $c \in \mathsf{C}$ that is honest in S, every identity $id \in \mathsf{ID}$ with $as = \mathsf{governor}(id)$ being an honest AS and with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S, every service session identified by some nonce n for id at c, n is not derivable from the attackers knowledge in S (i.e., $n \notin d_{\emptyset}(S(\mathsf{attacker}))$).

C. Session Integrity for Authentication and Authorization

In addition to the authorization and authentication properties, it is important that the integrity of user sessions is not compromised. This is captured by two different session integrity properties. The first one, session integrity for authorization, ensures that an honest user should never use resources of the attacker. The second property, session integrity for authentication, captures that an honest user should never be logged in under the identity of the attacker.

We first define notations for the processing steps that represent important events during a flow of a FAPI web system.

Definition 12 (User is logged in). For a run ρ of a FAPI web system with a network attacker $\mathcal{FAPI}$ we say that a browser b was authenticated to a client c using an authorization server as and an identity u in a login session identified by a nonce $lsid$ in processing step Q in ρ with

$$Q = (S, E, N) \xrightarrow[r \to E_{\text{out}}]{} (S', E', N')$$

(for some S, S', E, E', N, N') and some event $\langle y, y', m\rangle \in E_{\text{out}}$ such that m is an HTTPS response matching an HTTPS request sent by b to c and we have that in the headers of m there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\langle \mathtt{\_\_Host}, \mathtt{serviceSessionId}\rangle{:}\langle ssid, \top, \top, \top\rangle]\rangle$ for some nonce $ssid$ such that $S(c).\mathtt{sessions}[lsid][\mathtt{serviceSessionId}] \equiv ssid$ and $S(c).\mathtt{sessions}[lsid][\mathtt{loggedInAs}] \equiv \langle d, u\rangle$ with $d \in \mathsf{dom}(as)$. We then write $\mathsf{loggedIn}_{\rho}^{Q}(b, c, u, as, lsid)$.

Definition 13 (User started a login flow). For a run ρ of a FAPI web system with a network attacker $\mathcal{FAPI}$ we say that the user of the browser b started a login session identified by a nonce $lsid$ at the client c in a processing step Q in ρ if (1) in that processing step, the browser b was triggered, selected a document loaded from an origin of c, executed the script $script\_client\_index$ in that document, and in that script, executed the Line 7 of Algorithm 8, and (2) c sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle \mathtt{Set\text{-}Cookie}, [\langle \mathtt{\_\_Host}, \mathtt{sessionId}\rangle{:}\langle lsid, \top, \top, \top\rangle]\rangle$. We then write $\mathsf{started}_{\rho}^{Q}(b, c, lsid)$.

Definition 14 (User authenticated at an AS). For a run ρ of a FAPI web system with a network attacker $\mathcal{FAPI}$ we say that the user of the browser b authenticated to an authorization server as using an identity u for a login session identified by a nonce $lsid$ at the client c if there is a processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N') in which the browser b was triggered, selected a document loaded from an origin of as, executed the script $script\_as\_form$ in that document, and in that script, (1) in Line 4 of Algorithm 12, selected the identity u, and (2) we have that the $scriptstate$ of that document, when triggered, contains a nonce s such that $scriptstate[\mathtt{state}] \equiv s$ and $S(r).\mathtt{sessions}[lsid][\mathtt{state}] \equiv s$. We then write $\mathsf{authenticated}_{\rho}^{Q}(b, c, u, as, lsid)$.

Definition 15 (Resource Access). For a run ρ of a FAPI web system with a network attacker $\mathcal{FAPI}$ we say that a browser b accesses a resource of identity u stored at resource server rs through the session of client c identified by the nonce $lsid$ in processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N') with (1) $\langle\langle \mathtt{\_\_Host}, \mathtt{sessionid}\rangle, \langle lsid, y, z, z'\rangle\rangle \in^{\langle\rangle} S(b).\mathtt{cookies}[d]$ for $d \in \mathsf{dom}(c)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$, (2) $S(c).\mathtt{sessions}[lsid][\mathtt{resource}] \equiv r$ with $r \in s_0^{rs}.\mathtt{resourceNonce}[u]$ and (3) $S(c).\mathtt{sessions}[lsid][\mathtt{resource\_server}] \in \mathsf{dom}(rs)$. We then write $\mathsf{accessesResource}_{\rho}^{Q}(b, r, u, c, rs, lsid)$.

Session Integrity Property for Authentication

This security property captures that (a) a user should only be logged in when the user actually expressed the wish to start a FAPI flow before, and (b) if a user expressed the wish to start a FAPI flow using some honest authorization server and a specific identity, then user is not logged in under a different identity.

Definition 16 (Session Integrity for Authentication). Let $\mathcal{FAPI}$ be an FAPI web system with a network attacker. We say that *$\mathcal{FAPI}$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{FAPI}$, every processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client $c \in \mathsf{C}$ that is honest in S, every nonce $lsid$, and $\mathsf{loggedIn}_\rho^Q(b, c, u, as, lsid)$ we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.

Session Integrity Property for Authorization

This security property captures that (a) a user should only access resources when the user actually expressed the wish to start a FAPI flow before, and (b) if a user expressed the wish to start a FAPI flow using some honest authorization server and a specific identity, then user is not using resources of a different identity. We note that for this, we require that the resource server which the client uses is honest, as otherwise, the attacker can trivially return any resource.

Definition 17 (Session Integrity for Authorization). Let $\mathcal{FAPI}$ be a FAPI web system with a network attackers. We say that *$\mathcal{FAPI}$ is secure w.r.t. session integrity for authorization* iff for every run ρ of $\mathcal{FAPI}$, every processing step Q in ρ with

$$Q = (S, E, N) \to (S', E', N')$$

(for some S, S', E, E', N, N'), every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity u, every client $c \in \mathsf{C}$ that is honest in S, every $rs \in \mathsf{RS}$ that is honest in S, every nonce r, every nonce $lsid$, we have that if $\mathsf{accessesResource}_\rho^{\mathsf{Q}}(\mathsf{b}, \mathsf{r}, \mathsf{u}, \mathsf{c}, rs, lsid)$ and $s_0^{rs}.\mathtt{authServ} \in \mathsf{dom}(as)$, then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.

By *session integrity* we denote the conjunction of both properties.

References

[1] R. Berjon et al., eds. *HTML5, W3C Recommendation*. 2014. URL: http://www.w3.org/TR/html5/.

[2] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. 2020. URL: https://www.rfc-editor.org/info/rfc8705.

[3] L. Chen, S. Englehardt, M. West, and J. Wilander. *Cookies: HTTP State Management Mechanism*. Internet-Draft draft-ietf-httpbis-rfc6265bis-09. Work in Progress. Internet Engineering Task Force, 2021. 59 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-09.

[4] D. Fett. "An Expressive Formal Model of the Web Infrastructure". PhD thesis. 2018.

[5] D. Fett. *FAPI 2.0 Attacker Model, Commit 209f58a*. OpenID Foundation. Jun. 1, 2022. https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Attacker_Model.md. 2022.

[6] D. Fett. *FAPI 2.0 Baseline Profile, Commit 209f58a*. OpenID Foundation. Jun. 1, 2022. https://bitbucket.org/openid/fapi/src/209f58afbd41fb20ab3ed65ca4e2f67ffd5dda77/FAPI_2_0_Baseline_Profile.md. 2022.

[7] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite. *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)*. Internet-Draft draft-ietf-oauth-dpop-08. Work in Progress. Internet Engineering Task Force, 2022. 41 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-dpop/08/.

[8] D. Fett, P. Hosseyni, and R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-grade API". In: *IEEE S&P*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 1054–1072.

[9] D. Fett, P. Hosseyni, and R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-grade API". In: *CoRR* abs/1901.11520 (2019). arXiv: 1901.11520. URL: http://arxiv.org/abs/1901.11520.

[10] D. Fett, R. Küsters, and G. Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System". In: *IEEE S&P*. IEEE Computer Society, 2014, pp. 673–688.

[11] D. Fett, R. Küsters, and G. Schmitz. "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web". In: *ESORICS*. Vol. 9326. LNCS. Springer, 2015, pp. 43–65.

[12] D. Fett, R. Küsters, and G. Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *ACM CCS*. ACM, 2015, pp. 1358–1369.

[13] D. Fett, R. Küsters, and G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *ACM CCS*. ACM, 2016, pp. 1204–1215.

[14] D. Fett, R. Küsters, and G. Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *CSF*. IEEE Computer Society, 2017.

[15] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. 2012. URL: https://www.rfc-editor.org/info/rfc6749.

[16] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. 2012. URL: https://www.rfc-editor.org/info/rfc6750.

[17] M. Jones, N. Sakimura, and J. Bradley. *OAuth 2.0 Authorization Server Metadata*. RFC 8414. 2018. URL: https://www.rfc-editor.org/info/rfc8414.

[18] R. Küsters, G. Schmitz, and D. Fett. *The Web Infrastructure Model (WIM)*. Technical Report. Version 1.0. 2022. URL: https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf.

[19] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. *OAuth 2.0 Security Best Current Practice*. Internet-Draft. Work in Progress. Internet Engineering Task Force, 2021. 52 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/19/.

[20] T. Lodderstedt, B. Campbell, N. Sakimura, D. Tonge, and F. Skokan. *OAuth 2.0 Pushed Authorization Requests*. RFC 9126. 2021. URL: https://www.rfc-editor.org/info/rfc9126.

[21] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. 2015. URL: https://www.rfc-editor.org/info/rfc7617.

[22] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008. URL: https://www.rfc-editor.org/info/rfc5246.

[23] J. Richer. *OAuth 2.0 Token Introspection*. RFC 7662. 2015. URL: https://www.rfc-editor.org/info/rfc7662.

[24] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.

[25] N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. 2015. URL: https://www.rfc-editor.org/info/rfc7636.

[26] K. M. zu Selhausen and D. Fett. *OAuth 2.0 Authorization Server Issuer Identification*. RFC 9207. 2022. URL: https://www.rfc-editor.org/info/rfc9207.

Appendix A
Technical Definitions

Here, we provide technical definitions of the WIM. These follow the descriptions in [4, 8, 10–14]. We refer the reader to [18] for a full reference version of the WIM.

A. Terms and Notations

Definition 18 (Nonces and Terms). By $X = \{x_0, x_1, \dots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol for some $n \geq 0$ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ (see Figure 4). For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 19 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \dots\}$ of variables (called placeholders). The set $\mathcal{M}^\nu := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of *protomessages*, i.e., messages that can contain placeholders.

Example 1. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k))$ is interpreted to be the message $\langle a, b, c\rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

Definition 20 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m\rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \dots\rangle$, etc.).

Definition 21 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 4. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 22 (Pattern Matching). Let $\mathit{pattern} \in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches* $\mathit{pattern}$ iff t can be acquired from $\mathit{pattern}$ by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim \mathit{pattern}$. For a sequence of patterns $\mathit{patterns}$ we write $t \mathrel{\dot{\sim}} \mathit{patterns}$ to denote that t matches at least one pattern in $\mathit{patterns}$.

For a term t' we write $t'|\,\mathit{pattern}$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match $\mathit{pattern}$.

Example 2. For example, for a pattern $p = \langle \top, *\rangle$ we have that $\langle \top, 42\rangle \sim p$, $\langle \bot, 42\rangle \not\sim p$, and

$$\langle\langle \bot, \top\rangle, \langle \top, 23\rangle, \langle \mathsf{a}, \mathsf{b}\rangle, \langle \top, \bot\rangle\rangle|\,p = \langle\langle \top, 23\rangle, \langle \top, \bot\rangle\rangle\ .$$

Definition 23 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \dots, x_n\})$, and $t_1, \dots, t_n \in \mathcal{T}_N$.

By $\tau[t_1/x_1, \dots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \dots, n\}$.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{1}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{2}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{3}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{4}$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \tag{5}$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \tag{6}$$
$$\pi_i(\langle x_1, \dots, x_n\rangle) = x_i \ \text{ if } 1 \leq i \leq n \tag{7}$$
$$\pi_j(\langle x_1, \dots, x_n\rangle) = \Diamond \ \text{ if } j \notin \{1, \dots, n\} \tag{8}$$
$$\pi_j(t) = \Diamond \ \text{ if } t \text{ is not a sequence} \tag{9}$$

Figure 4. Equational theory for Σ.

Definition 24 (Sequence Notations). For a sequence $t = \langle t_1, \dots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \dots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \dots, t_n, y \rangle$. For a sequence $r = \langle r_1, \dots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \dots, t_n, r_1, \dots, r_m \rangle$. We define $|t| = n$. If t is not a sequence, we set $|t| = \diamond$. For a finite set M with $M = \{m_1, \dots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \dots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.)

Definition 25 (Dictionaries). A *dictionary over X and Y* is a term of the form

$$\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \dots, k_n \in X$, $v_1, \dots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \dots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$.

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure 5 shows the short notation for dictionary operations. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \dots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. (Note that if a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply non-deterministically to one of both elements.) If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \dots, k_i : v_i, \dots, k_n : v_n]\,[k_i] = v_i \tag{10}$$

$$\begin{aligned}[k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \dots, k_n : v_n] - k_i = \\ [k_1 : v_1, \dots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \dots, k_n : v_n]\end{aligned} \tag{11}$$

Figure 5. Dictionary operators with $1 \leq i \leq n$.

Given a term $t = \langle t_1, \dots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 26 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

Example 3. For the term $\tau = \langle a, b, \langle c, d, \langle e, f \rangle \rangle \rangle$ and the pointer $\overline{p} = \langle 3, 1 \rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

To improve readability, we try to avoid writing, e.g., $o.2$ or $\pi_2(o)$ in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as $\langle \mathit{host}, \mathit{protocol} \rangle$ and o is an Origin term, then we can write $o.\mathtt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Definition 27 (Concatenation of Sequences). For a sequence $a = \langle a_1, \dots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \dots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \dots, a_i, b_1, b_2, \dots \rangle$.

Definition 28 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

B. Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of the web model presented in the following.

1) URLs:

Definition 29. A *URL* is a term of the form

$$\langle \mathtt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment} \rangle$$

with $\mathit{protocol} \in \{\mathtt{P}, \mathtt{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), a domain $\mathit{host} \in \mathsf{Doms}$, $\mathit{path} \in \mathbb{S}$, $\mathit{parameters} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, and $\mathit{fragment} \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs.

The $\mathit{fragment}$ part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$. We sometimes also write $\mathsf{URL}^{\mathit{host}}_{\mathit{path}}$ to denote the URL $\langle \mathtt{URL}, \mathtt{S}, \mathit{host}, \mathit{path}, \langle\rangle, \bot \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 26):

Example 4. For the URL $u = \langle \texttt{URL}, a, b, c, d\rangle$, $u.\texttt{protocol} = a$. If, in the algorithms described later, we say $u.\texttt{path} := e$ then $u = \langle \texttt{URL}, a, b, c, e\rangle$ afterwards.

2) Origins:

Definition 30. An *origin* is a term of the form $\langle host, protocol\rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\texttt{P}, \texttt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

Example 5. For example, $\langle \texttt{FOO}, \texttt{S}\rangle$ is the HTTPS origin for the domain FOO, while $\langle \texttt{BAR}, \texttt{P}\rangle$ is the HTTP origin for the domain BAR.

3) Cookies:

Definition 31. A *cookie* is a term of the form $\langle name, content\rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and $content$ is a term of the form $\langle value, secure, session, httpOnly\rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, $secure$, $session$, $httpOnly \in \{\top, \bot\}$. As $name$ is a term, it may also be a sequence consisting of two parts. If the name consists of two parts, we call the first part of the sequence (i.e., $name.1$) the *prefix* of the name. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^\nu$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the $secure$ attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections.[5] If the $session$ flag is set, this cookie will be deleted as soon as the browser is closed. The $httpOnly$ attribute controls whether scripts have access to this cookie.

When the __Host prefix (see [3]) of a cookie is set (i.e., $name$ consists of two parts and $name.1 \equiv \texttt{\_\_Host}$), the browser accepts the cookie only if the $secure$ attribute is set. As such cookies are only transferred over secure channels (i.e., with TLS), the cookie cannot be set by a network attacker. Note that the WIM does not model the domain attribute of the Set-Cookie header, so cookies in the WIM are always sent to the originating domain and not some subdomain. Therefore, the WIM models only the __Host prefix, but not the __Secure prefix.

Also note that cookies of the form described here are only contained in HTTP(S) responses. In HTTP(S) requests, only the components $name$ and $value$ are transferred as a pairing of the form $\langle name, value\rangle$.

4) HTTP Messages:

Definition 32. An *HTTP request* is a term of the form shown in (12). An *HTTP response* is a term of the form shown in (13).

$$\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle \tag{12}$$
$$\langle \texttt{HTTPResp}, nonce, status, headers, body\rangle \tag{13}$$

The components are defined as follows:

- $nonce \in \mathcal{N}$ serves to map each response to the corresponding request.
- $method \in \mathsf{Methods}$ is one of the HTTP methods.
- $host \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $path \in \mathbb{S}$ indicates the resource path at the server side.
- $status \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard).
- $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters.
- $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \texttt{Origin}, o\rangle$ where o is an origin,
  - $\langle \texttt{Set-Cookie}, c\rangle$ where c is a sequence of cookies,
  - $\langle \texttt{Cookie}, c\rangle$ where $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \texttt{Location}, l\rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \texttt{Referer}, r\rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \texttt{Strict-Transport-Security}, \top\rangle$,
  - $\langle \texttt{Authorization}, \langle username, password\rangle\rangle$ where $username$, $password \in \mathbb{S}$ (this header models the 'Basic' HTTP Authentication Scheme, see [21]),
  - $\langle \texttt{ReferrerPolicy}, p\rangle$ where $p \in \{\texttt{noreferrer}, \texttt{origin}\}$.
- $body \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

We write $\mathsf{HTTPRequests}$/$\mathsf{HTTPResponses}$ for the set of all HTTP requests or responses, respectively.

[5] Note that $secure$ cookies can be set over unencrypted connections (c.f. RFC 6265).

Example 6 (HTTP Request and Response).

$$
\begin{aligned}
r :=& \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, \texttt{1}\rangle\rangle, \\
& [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S}\rangle], \langle \texttt{foo}, \texttt{bar}\rangle\rangle && (14) \\
s :=& \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x\rangle\rangle && (15)
\end{aligned}
$$

An HTTP `POST` request for the URL http://example.com/show?index=1 is shown in (14), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar}\rangle$. A possible response is shown in (15), which contains an httpOnly cookie with name `SID` and value n_2 as well as a string `somescript` representing a script that can later be executed in the browser (see Section A-K) and the scripts initial state x.

a) Encrypted HTTP Messages: For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supposed to encrypt the response using the symmetric key.

Definition 33. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k'\rangle, k)$, where $k \in \mathit{terms}$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

We say that an HTTP(S) response matches or corresponds to an HTTP(S) request if both terms contain the same nonce.

Example 7.

$$
\begin{aligned}
\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})) && (16) \\
\mathsf{enc_s}(s, k') && (17)
\end{aligned}
$$

The term (16) shows an encrypted request (with r as in (14)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (17) is a response (with s as in (15)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (16).

5) DNS Messages:

Definition 34. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, \mathit{domain}, \mathit{nonce}\rangle$ where $\mathit{domain} \in \mathsf{Doms}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 35. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, \mathit{domain}, \mathit{result}, \mathit{nonce}\rangle$ with $\mathit{domain} \in \mathsf{Doms}$, $\mathit{result} \in \mathsf{IPs}$, $\mathit{nonce} \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

C. Atomic Processes, Systems and Runs

Entities that take part in a network are modeled as atomic processes. An atomic process takes a term that describes its current state and an event as input, and then (non-deterministically) outputs a new state and a sequence of events.

Definition 36 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where $I^p \subseteq \mathsf{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^{\nu}\langle\rangle} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p. For any new state s and any sequence of nonces $(\eta_1, \eta_2, \dots)$ we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \dots] \in Z^p$. A *system* $\mathcal{P}$ is a (possibly infinite) set of atomic processes.

Definition 37 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[6] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

Definition 38 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

$$(S, E, N) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S', E', N')$$

where

[6] Here: Not in the sense of terms as defined earlier.

1) (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
2) $e_{\text{in}} = \langle a, f, m \rangle \in E$ is an event,
3) $p \in \mathcal{P}$ is a process,
4) E_{out} is a sequence (term) of events

such that there exists

1) a sequence (term) $E^{\nu}_{\text{out}} \subseteq 2^{\mathcal{E}^{\nu}\langle\rangle}$ of protoevents,
2) a term $s^{\nu} \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3) a sequence $(v_1, v_2, \ldots, v_i)$ of all placeholders appearing in E^{ν}_{out} (ordered lexicographically),
4) a sequence $N^{\nu} = (\eta_1, \eta_2, \ldots, \eta_i)$ of the first i elements in N

with

1) $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
2) $E_{\text{out}} = E^{\nu}_{\text{out}}[\eta_1/v_1, \ldots, \eta_i/v_i]$,
3) $S'(p) = s^{\nu}[\eta_1/v_1, \ldots, \eta_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$,
4) $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$,
5) $N' = N \setminus N^{\nu}$.

We may omit the superscript and/or subscript of the arrow.

Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any occurences of placeholders ν_x by "fresh" nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

Definition 39 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \rightarrow (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a finite run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 containing infinite trigger events of the form $\langle a, a, \texttt{TRIGGER} \rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

D. Atomic Dolev-Yao Processes

We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Definition 40 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_{\emptyset}(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, the term a can be derived from the set of terms $\{\mathsf{enc_a}(\langle a, b, c \rangle, \mathsf{pub}(k)), k\}$, i.e., $a \in d_{\emptyset}(\{\mathsf{enc_a}(\langle a, b, c \rangle, \mathsf{pub}(k)), k\})$.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be derived from the input event and the state.

Definition 41 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s^p_0)$ such that p is an atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that E, $s' \in d_{V_{\text{process}}}(\{e, s\})$.

E. Attackers

The so-called *attacker process* is a Dolev-Yao process which records all messages it receives and outputs any finite sequence of events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties (using corrupt messages).

Definition 42 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses $A \subseteq \mathsf{IPs}$* is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_{\mathcal{N}}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s \rangle$ and $E = \langle \langle a_1, f_1, m_1 \rangle, \ldots, \langle a_n, f_n, m_n \rangle \rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$.

Placeholder	Usage
ν_1	Algorithm 23, new window nonces
ν_2	Algorithm 23, new HTTP request nonce
ν_3	Algorithm 23, lookup key for pending HTTP requests entry
ν_4	Algorithm 21, new HTTP request nonce (multiple lines)
ν_5	Algorithm 21, new subwindow nonce
ν_6	Algorithm 22, new HTTP request nonce
ν_7	Algorithm 22, new document nonce
ν_8	Algorithm 18, lookup key for pending DNS entry
ν_9	Algorithm 15, new window nonce
$\nu_{10}, \dots$	Algorithm 21, replacement for placeholders in script output

Table II: List of placeholders used in browser algorithms.

Note that in a web system, we distinguish between two kinds of attacker processes: web attackers and network attackers. Both kinds match the definition above, but differ in the set of assigned addresses in the context of a web system. While for web attackers, the set of addresses I^p is disjoint from other web attackers and honest processes, i.e., web attackers participate in the network as any other party, the set of addresses I^p of a network attacker is not restricted. Hence, a network attacker can intercept events addressed to any party as well as spoof all addresses. Note that one network attacker subsumes any number of web attackers as well as any number of network attackers.

F. Notations for Functions and Algorithms

When describing algorithms, we use the following notations:

1) Non-deterministic choosing and iteration: The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set N. We write **for each** $s \in M$ **do** to denote that the following commands (until **end for**) are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We write, for example,

let x, y **such that** $\langle \texttt{Constant}, x, y \rangle \equiv t$ **if possible; otherwise** doSomethingElse

for some variables x, y, a string $\texttt{Constant}$, and some term t to express that $x := \pi_2(t)$, and $y := \pi_3(t)$ if $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y \rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

2) Function calls: When calling functions that do not return anything, we write

call $\mathsf{FUNCTION\_NAME}(x, y)$

to describe that a function $\mathsf{FUNCTION\_NAME}$ is called with two variables x and y as parameters. If that function executes the command **stop** E, s', the processing step terminates, where E is the sequence of events output by the associated process and s' is its new state. If that function does not terminate with a **stop**, the control flow returns to the calling function at the next line after the call.

When calling a function that has a return value, we omit the **call** and directly write

let $z := \mathsf{FUNCTION\_NAME}(x, y)$

to assign the return value to a variable z after the function returns. Note that the semantics for execution of **stop** within such functions is the same as for functions without a return value.

3) Stop without output: We write **stop** (without further parameters) to denote that there is no output and no change in the state.

4) Placeholders: In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 18). Table II shows a list of all placeholders used.

5) Abbreviations for URLs and Origins: We sometimes use an abbreviation for URLs. We write URL^{d}_{path} to describe the following URL term: $\langle \texttt{URL}, \texttt{S}, d, \mathit{path}, \langle\rangle \rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write $\mathsf{URL}^{\mathsf{P}}_{path}$. For a (secure) origin $\langle d, \texttt{S} \rangle$ of some domain d, we also write origin_d. Again, if the domain d belongs to some distinguished process P and d is the only domain associated to this process, we may write $\mathsf{origin}_{\mathsf{P}}$.

G. Browsers

Here, we present the formal model of browsers.

1) Scripts: Recall that a *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below).

Definition 43 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 44 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last scriptstate and limited information about the browser's state) s. The script then outputs a term s', which represents the new scriptstate and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \dots$ which will be replaced by nonces once the browser DY process finishes (effectively providing the script with a way to get "fresh" nonces).

Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

Definition 45 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

2) Web Browser State: Before we can define the state of a web browser, we first have to define windows and documents.

Definition 46. A *window* is a term of the form $w = \langle \mathit{nonce}, \mathit{documents}, \mathit{opener} \rangle$ with $\mathit{nonce} \in \mathcal{N}$, $\mathit{documents} \subset^{\langle\rangle} \mathsf{Documents}$ (defined below), $\mathit{opener} \in \mathcal{N} \cup \{\bot\}$ where $d.\mathtt{active} = \top$ for exactly one $d \in^{\langle\rangle} \mathit{documents}$ if $\mathit{documents}$ is not empty (we then call d the *active document of* w). We write $\mathsf{Windows}$ for the set of all windows. We write $w.\mathtt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

We will refer to the window nonce as *(window) reference*.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the "back" button) and the documents in the window term to the right of the currently active document are documents available via the "forward" button.

A window a may have opened a top-level window b (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term b is the nonce of a, i.e., $b.\mathtt{opener} = a.\mathtt{nonce}$.

Definition 47. A *document* d is a term of the form

$$\langle \mathit{nonce}, \mathit{location}, \mathit{headers}, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \mathit{scriptinputs}, \mathit{subwindows}, \mathit{active} \rangle$$

where $\mathit{nonce} \in \mathcal{N}$, $\mathit{location} \in \mathsf{URLs}$, $\mathit{headers} \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{referrer} \in \mathsf{URLs} \cup \{\bot\}$, $\mathit{script} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptstate} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptinputs} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{subwindows} \subset^{\langle\rangle} \mathsf{Windows}$, $\mathit{active} \in \{\top, \bot\}$. A *limited document* is a term of the form $\langle \mathit{nonce}, \mathit{subwindows} \rangle$ with nonce, $\mathit{subwindows}$ as above. A window $w \in^{\langle\rangle} \mathit{subwindows}$ is called a *subwindow* (of d). We write $\mathsf{Documents}$ for the set of all documents. For a document term d we write $d.\mathtt{origin}$ to denote the origin of the document, i.e., the term $\langle d.\mathtt{location}.\mathtt{host}, d.\mathtt{location}.\mathtt{protocol} \rangle \in \mathsf{Origins}$.

We will refer to the document nonce as *(document) reference*.

Definition 48. For two window terms w and w' we write

$$w \xrightarrow{\text{childof}} w'$$

if $w \in^{\langle\rangle} w'.\mathtt{activedocument}.\mathtt{subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure and we write $\xrightarrow{\text{childof}^*}$ for the reflexive transitive closure.

In the web browser state, HTTP(S) messages are tracked using *references*, where we distinguish between references for XMLHttpRequests and references for normal HTTP(S) requests.

Definition 49. A reference for a normal HTTP(S) request is a sequence of the form $\langle \mathtt{REQ}, \mathit{nonce} \rangle$, where nonce is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \mathtt{XHR}, \mathit{nonce}, \mathit{xhrreference} \rangle$, where nonce is a document reference and $\mathit{xhrreference}$ is some nonce that was chosen by the script that initiated the request.

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 25.

Definition 50. The *set of states* $Z_{\text{webbrowser}}$ *of a web browser atomic Dolev-Yao process* consists of the terms of the form

$$\langle \mathit{windows}, \mathit{ids}, \mathit{secrets}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{keyMapping}, \\ \mathit{sts}, \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{isCorrupted} \rangle$$

with the subterms as follows:

- $\mathit{windows} \subset^{\langle\rangle} \mathsf{Windows}$ contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).

- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ contains a list of secrets that are associated with certain origins (i.e., passwords of the user of the browser at certain websites). Note that this structure allows to have a single secret under an origin or a list of secrets under an origin.
- $cookies$ is a dictionary over Doms and sequences of Cookies modeling cookies that are stored for specific domains.
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle \,|\, o \in \mathsf{Origins},\, r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ maps domains to TLS encryption keys.
- $sts \subset^{\langle\rangle} \mathsf{Doms}$ stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in \mathsf{IPs}$ defines the IP address of the DNS server.
- $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url\rangle$. In these pairings, $reference$ is an HTTP(S) request reference (as above), $request$ contains the HTTP(S) message that awaits DNS resolution, and url contains the URL of said HTTP request. The pairings in $pendingDNS$ are indexed by the DNS request/response nonce.
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f\rangle$ with $reference$, $request$, and url as in $pendingDNS$, key is the symmetric encryption key if HTTPS is used or $\perp$ otherwise, and f is the IP address of the server to which the request was sent.
- $isCorrupted \in \{\perp, \mathtt{FULLCORRUPT}, \mathtt{CLOSECORRUPT}\}$ specifies the corruption level of the browser.

In corrupted browsers, certain subterms are used in different ways (e.g., $pendingRequests$ is used to store all observed messages).

3) Web Browser Relation: We will now define the relation $R_{\text{webbrowser}}$ of a standard HTTP browser. We first introduce some notations and then describe the functions that are used for defining the browser main algorithm. We then define the browser relation.

a) Helper Functions: In the following description of the web browser relation $R_{\text{webbrowser}}$ we use the helper functions Subwindows, Docs, Clean, CookieMerge, AddCookie, and NavigableWindows.

Subwindows and Docs. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[7] to windows in the window list $s.\mathtt{windows}$ and (recursively) the subwindows of their active documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 51. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\mathtt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

Given a browser state s, the set $\mathsf{Docs}(s)$ of pointers to active documents is the minimal set such that for every $\overline{p} \in \mathsf{Subwindows}(s)$ with $s.\overline{p}.\mathtt{activedocument} \not\equiv \langle\rangle$, there exists a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\mathtt{activedocument}$.

By $\mathsf{Subwindows}^+(s)$ and $\mathsf{Docs}^+(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

Clean. The function Clean will be used to determine which information about windows and documents the script running in the document d has access to.

Definition 52. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\mathtt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list, and (3) the values of the subterms $\mathtt{headers}$ for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

CookieMerge. The function CookieMerge merges two sequences of cookies together: When used in the browser, $oldcookies$ is the sequence of existing cookies for some origin, $newcookies$ is a sequence of new cookies that was output by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in RFC6265.

Definition 53. For a sequence of cookies (with pairwise different names) $oldcookies$, a sequence of cookies $newcookies$, and a string $protocol \in \{\mathsf{P}, \mathsf{S}\}$, the set $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is defined by the following

[7]Recall the definition of a pointer in Definition 26.

Algorithm 15 Web Browser Model: Determine window for navigation.

function GETNAVIGABLEWINDOW($\overline{w}$, $window$, $noreferrer$, s')
  if $window \equiv$ `_BLANK` **then** → Open a new window when `_BLANK` is used
    if $noreferrer \equiv \top$ **then**
      let $w' := \langle \nu_9, \langle\rangle, \bot \rangle$
    else
      let $w' := \langle \nu_9, \langle\rangle, s'.\overline{w}.\mathtt{nonce} \rangle$
    let $s'.\mathtt{windows} := s'.\mathtt{windows} +^{\langle\rangle} w'$
    ↪ **and** let $\overline{w'}$ be a pointer to this new element in s'
    return $\overline{w'}$
  let $\overline{w'} \leftarrow$ NavigableWindows($\overline{w}, s'$) **such that** $s'.\overline{w'}.\mathtt{nonce} \equiv window$
  ↪ **if possible; otherwise return** $\overline{w}$
  return $\overline{w'}$

Algorithm 16 Web Browser Model: Determine same-origin window.

function GETWINDOW($\overline{w}$, $window$, s')
  let $\overline{w'} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w'}.\mathtt{nonce} \equiv window$
  ↪ **if possible; otherwise return** $\overline{w}$
  if $s'.\overline{w'}.\mathtt{activedocument.origin} \equiv s'.\overline{w}.\mathtt{activedocument.origin}$ **then**
    return $\overline{w'}$
  return $\overline{w}$

algorithm: From $newcookies$ remove all cookies c that have $c.\mathtt{content.httpOnly} \equiv \top$ or where $(c.\mathtt{name}.1 \equiv \mathtt{\_\_Host}) \wedge ((protocol \equiv \mathtt{P}) \vee (c.\mathtt{secure} \equiv \bot))$. For any $c, c' \in^{\langle\rangle} newcookies$, $c.\mathtt{name} \equiv c'.\mathtt{name}$, remove the cookie that appears left of the other in $newcookies$. Let m be the set of cookies that have a name that either appears in $oldcookies$ or in $newcookies$, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle} oldcookies$, $c_{\text{new}} \in^{\langle\rangle} newcookies$, $c_{\text{old}}.\mathtt{name} \equiv c_{\text{new}}.\mathtt{name}$, add c_{new} to m if $c_{\text{old}}.\mathtt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is m.

AddCookie. The function AddCookie adds a cookie c received in an HTTP response to the sequence of cookies contained in the sequence $oldcookies$. It is again based on the algorithm described in RFC6265 but simplified for the use in the browser model.

Definition 54. For a sequence of cookies (with pairwise different names) $oldcookies$, a cookie c, and a string $protocol \in \{\mathtt{P}, \mathtt{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence $\mathsf{AddCookie}(oldcookies, c, protocol)$ is defined by the following algorithm: Let $m := oldcookies$. If $(c.\mathtt{name}.1 \equiv \mathtt{\_\_Host}) \wedge \neg((protocol \equiv \mathtt{S}) \wedge (c.\mathtt{secure} \equiv \top))$, then return m, else: Remove any c' from m that has $c.\mathtt{name} \equiv c'.\mathtt{name}$. Append c to m and return m.

NavigableWindows. The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [1], Section 5.1.4 for this definition.

Definition 55. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w'}.\mathtt{activedocument.origin} \equiv s'.\overline{w}.\mathtt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists\, \overline{w''} \in \mathsf{Subwindows}(s')$ with $s'.\overline{w'} \xrightarrow{\text{childof}^*} s'.\overline{w''}$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\, s'.\overline{p}.\mathtt{activedocument.origin} = s'.\overline{w}.\mathtt{activedocument.origin}$ ($\overline{w'}$ is not a top-level window but there is an ancestor window $\overline{p}$ of $\overline{w'}$ with an active document that has the same origin as the active document in $\overline{w}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'}.\mathtt{opener} = s'.\overline{p}.\mathtt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ is a top-level window—it has an opener—and $\overline{w}$ is allowed to navigate the opener window of $\overline{w'}$, $\overline{p}$), then $\overline{w'} \in \overline{W}$.

b) Functions:

- The function GETNAVIGABLEWINDOW (Algorithm 15) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When

Algorithm 17 Web Browser Model: Cancel pending requests for given window.

function $\mathsf{CANCELNAV}(\mathit{reference}, s')$
  remove all $\langle \mathit{reference}, \mathit{req}, \mathit{url}, \mathit{key}, f\rangle$ **from** $s'.\mathtt{pendingRequests}$ **for any** req, url, key, f
  remove all $\langle x, \langle \mathit{reference}, \mathit{message}, \mathit{url}\rangle\rangle$ **from** $s'.\mathtt{pendingDNS}$
  $\hookrightarrow$ **for any** x, $\mathit{message}$, url
  return s'

Algorithm 18 Web Browser Model: Prepare headers, do DNS resolution, save message.

function $\mathsf{HTTP\_SEND}(\mathit{reference}, \mathit{message}, \mathit{url}, \mathit{origin}, \mathit{referrer}, \mathit{referrerPolicy}, a, s')$
  if $\mathit{message}.\mathtt{host} \in^{\langle\rangle} s'.\mathtt{sts}$ **then**
    let $\mathit{url}.\mathtt{protocol} := \mathtt{S}$
  let $\mathit{cookies} := \langle\{\langle c.\mathtt{name}, c.\mathtt{content}.\mathtt{value}\rangle \mid c \in^{\langle\rangle} s'.\mathtt{cookies}[\mathit{message}.\mathtt{host}]$
  $\hookrightarrow \wedge (c.\mathtt{content}.\mathtt{secure} \equiv \top \implies (\mathit{url}.\mathtt{protocol} \equiv \mathtt{S}))\}\rangle$
  let $\mathit{message}.\mathtt{headers}[\mathtt{Cookie}] := \mathit{cookies}$
  if $\mathit{origin} \not\equiv \bot$ **then**
    let $\mathit{message}.\mathtt{headers}[\mathtt{Origin}] := \mathit{origin}$
  if $\mathit{referrerPolicy} \equiv \mathtt{no\text{-}referrer}$ **then**
    let $\mathit{referrer} := \bot$
  if $\mathit{referrer} \not\equiv \bot$ **then**
    if $\mathit{referrerPolicy} \equiv \mathtt{origin}$ **then**
      let $\mathit{referrer} := \langle \mathtt{URL}, \mathit{referrer}.\mathtt{protocol}, \mathit{referrer}.\mathtt{host}, /, \langle\rangle, \bot\rangle$
        $\rightarrow$ Referrer stripped down to origin.
    let $\mathit{referrer}.\mathtt{fragment} := \bot$
      $\rightarrow$ Browsers do not send fragment identifiers in the Referer header.
    let $\mathit{message}.\mathtt{headers}[\mathtt{Referer}] := \mathit{referrer}$
  let $s'.\mathtt{pendingDNS}[\nu_8] := \langle \mathit{reference}, \mathit{message}, \mathit{url}\rangle$
  stop $\langle\langle s'.\mathtt{DNSaddress}, a, \langle \mathtt{DNSResolve}, \mathit{message}.\mathtt{host}, \nu_8\rangle\rangle\rangle$, s'

it is given a window reference (nonce) window, this function returns a pointer to a selected window term in s':

- If window is the string $\mathtt{\_BLANK}$, a new window is created and a pointer to that window is returned.
- If window is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by $\mathsf{NavigableWindows}$ above).

In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).

- The function $\mathsf{GETWINDOW}$ (Algorithm 16) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.
- The function $\mathsf{CANCELNAV}$ (Algorithm 17) is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference.
- The function $\mathsf{HTTP\_SEND}$ (Algorithm 18) takes an HTTP request $\mathit{message}$ as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in $s'.\mathtt{pendingDNS}$ until

Algorithm 19 Web Browser Model: Navigate a window backward.

function $\mathsf{NAVBACK}(\overline{w'}, s')$
  if $\exists \overline{j} \in \mathbb{N}, \overline{j} > 1$ **such that** $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} \equiv \top$ **then**
    let $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} := \bot$
    let $s'.\overline{w'}.\mathtt{documents}.(\overline{j} - 1).\mathtt{active} := \top$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w'}.\mathtt{nonce}, s')$
  stop $\langle\rangle$, s'

Algorithm 20 Web Browser Model: Navigate a window forward.

function $\mathsf{NAVFORWARD}(\overline{w'}, s')$
  if $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} \equiv \top$
    $\hookrightarrow \wedge\ s'.\overline{w'}.\mathtt{documents}.(\overline{j} + 1) \in \mathsf{Documents}$ **then**
    let $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} := \bot$
    let $s'.\overline{w'}.\mathtt{documents}.(\overline{j} + 1).\mathtt{active} := \top$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w'}.\mathtt{nonce}, s')$
  stop $\langle\rangle$, s'

Algorithm 21 Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')

  let $\mathit{tree} := \mathsf{Clean}(s', s'.\overline{d})$

  let $\mathit{cookies} := \langle\{\langle c.\mathtt{name}, c.\mathtt{content.value}\rangle | c \in^{\langle\rangle} s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right]$
  $\hookrightarrow\ \wedge\ c.\mathtt{content.httpOnly} \equiv \bot$
  $\hookrightarrow\ \wedge\ \big(c.\mathtt{content.secure} \equiv \top \implies \big(s'.\overline{d}.\mathtt{origin.protocol} \equiv \mathtt{S}\big)\big)\}\rangle$

  let $\mathit{tlw} \leftarrow s'.\mathtt{windows}$ **such that** tlw is the top-level window containing $\overline{d}$

  let $\mathit{sessionStorage} := s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, \mathit{tlw}.\mathtt{nonce}\rangle\right]$

  let $\mathit{localStorage} := s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right]$

  let $\mathit{secrets} := s'.\mathtt{secrets}\left[s'.\overline{d}.\mathtt{origin}\right]$

  let $R := \mathsf{script}^{-1}(s'.\overline{d}.\mathtt{script})$ **if possible; otherwise stop**

  let $\mathit{in} := \langle \mathit{tree}, s'.\overline{d}.\mathtt{nonce}, s'.\overline{d}.\mathtt{scriptstate}, s'.\overline{d}.\mathtt{scriptinputs}, \mathit{cookies},$
  $\hookrightarrow\ \mathit{localStorage}, \mathit{sessionStorage}, s'.\mathtt{ids}, \mathit{secrets}\rangle$

  let $\mathit{state}' \leftarrow \mathscr{T}_{\mathcal{N}}(V_{\text{process}})$, $\mathit{cookies}' \leftarrow \mathsf{Cookies}^{\nu}$, $\mathit{localStorage}' \leftarrow \mathscr{T}_{\mathcal{N}}(V_{\text{process}})$,
  $\hookrightarrow\ \mathit{sessionStorage}' \leftarrow \mathscr{T}_{\mathcal{N}}(V_{\text{process}})$, $\mathit{command} \leftarrow \mathscr{T}_{\mathcal{N}}(V_{\text{process}})$,
  $\hookrightarrow\ \mathit{out} := \langle \mathit{state}', \mathit{cookies}', \mathit{localStorage}', \mathit{sessionStorage}', \mathit{command}\rangle$
  $\hookrightarrow$ **such that** $\mathit{out} := \mathit{out}^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$ **with** $(\mathit{in}, \mathit{out}^{\lambda}) \in R$

  let $s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right] :=$
  $\hookrightarrow\ \langle \mathsf{CookieMerge}(s'.\mathtt{cookies}\left[s'.\overline{d}.\mathtt{origin.host}\right], \mathit{cookies}', s'.\overline{d}.\mathtt{origin.protocol})\rangle$

  let $s'.\mathtt{localStorage}\left[s'.\overline{d}.\mathtt{origin}\right] := \mathit{localStorage}'$

  let $s'.\mathtt{sessionStorage}\left[\langle s'.\overline{d}.\mathtt{origin}, \mathit{tlw}.\mathtt{nonce}\rangle\right] := \mathit{sessionStorage}'$

  let $s'.\overline{d}.\mathtt{scriptstate} := \mathit{state}'$

  let $\mathit{referrer} := s'.\overline{d}.\mathtt{location}$

  let $\mathit{referrerPolicy} := s'.\overline{d}.\mathtt{headers}[\mathtt{ReferrerPolicy}]$

  let $\mathit{docorigin} := s'.\overline{d}.\mathtt{origin}$

  switch $\mathit{command}$ **do**

    case $\langle \mathtt{HREF}, \mathit{url}, \mathit{hrefwindow}, \mathit{noreferrer}\rangle$

      let $\overline{w'} :=$ GETNAVIGABLEWINDOW($\overline{w}$, $\mathit{hrefwindow}$, $\mathit{noreferrer}$, s')

      let $\mathit{reference} := \langle \mathtt{REQ}, s'.\overline{w'}.\mathtt{nonce}\rangle$

      let $\mathit{req} := \langle \mathtt{HTTPReq}, \nu_4, \mathtt{GET}, \mathit{url}.\mathtt{host}, \mathit{url}.\mathtt{path}, \mathit{url}.\mathtt{parameters}, \langle\rangle, \langle\rangle\rangle$

      if $\mathit{noreferrer} \equiv \top$ **then**

        let $\mathit{referrerPolicy} := \mathtt{noreferrer}$

      let $s' :=$ CANCELNAV($\mathit{reference}$, s')

      call HTTP_SEND($\mathit{reference}$, req, url, $\bot$, $\mathit{referrer}$, $\mathit{referrerPolicy}$, a, s')

    case $\langle \mathtt{IFRAME}, \mathit{url}, \mathit{window}\rangle$

      if $\mathit{window} \equiv \mathtt{\_SELF}$ **then**

        let $\overline{w'} := \overline{w}$

      else

        let $\overline{w'} :=$ GETWINDOW($\overline{w}$, window, s')

      let $\mathit{req} := \langle \mathtt{HTTPReq}, \nu_4, \mathtt{GET}, \mathit{url}.\mathtt{host}, \mathit{url}.\mathtt{path}, \mathit{url}.\mathtt{parameters}, \langle\rangle, \langle\rangle\rangle$

      let $w' := \langle \nu_5, \langle\rangle, \bot\rangle$

      let $s'.\overline{w'}.\mathtt{activedocument.subwindows} := s'.\overline{w'}.\mathtt{activedocument.subwindows} +^{\langle\rangle} w'$

      call HTTP_SEND($\langle \mathtt{REQ}, \nu_5\rangle$, req, url, $\bot$, $\mathit{referrer}$, $\mathit{referrerPolicy}$, a, s')

---

the DNS resolution finishes. *reference* is a reference as defined in Definition 49. *url* contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

- The functions NAVBACK (Algorithm 19) and NAVFORWARD (Algorithm 20), navigate a window backward or forward. More precisely, they deactivate one document and activate that document's preceding document or succeeding document, respectively. If no such predecessor/successor exists, the functions do not change the state.
- The function RUNSCRIPT (Algorithm 21) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.
- The function PROCESSRESPONSE (Algorithm 22) is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. *reference* is a reference as defined in Definition 49. *requestUrl* contains the URL used when retrieving the document.
  The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response to the respective receiver (for XHR responses).

c) Browser Relation: We can now define the *relation $R_{\text{webbrowser}}$ of a web browser atomic process* as follows:

case $\langle \mathtt{FORM}, url, method, data, hrefwindow \rangle$
  if $method \not\in \{\mathtt{GET}, \mathtt{POST}\}$ **then**
    stop
  let $\overline{w'} := \mathsf{GETNAVIGABLEWINDOW}(\overline{w},\ hrefwindow,\ \bot,\ s')$
  let $reference := \langle \mathtt{REQ}, s'.\overline{w'}.\mathtt{nonce} \rangle$
  if $method = \mathtt{GET}$ **then**
    let $body := \langle\rangle$
    let $parameters := data$
    let $origin := \bot$
  else
    let $body := data$
    let $parameters := url.\mathtt{parameters}$
    let $origin := docorigin$
  let $req := \langle \mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, parameters, \langle\rangle, body \rangle$
  let $s' := \mathsf{CANCELNAV}(reference, s')$
  call $\mathsf{HTTP\_SEND}(reference,\ req,\ url,\ origin,\ referrer,\ referrerPolicy,\ a,\ s')$
case $\langle \mathtt{SETSCRIPT}, window, script \rangle$
  let $\overline{w'} := \mathsf{GETWINDOW}(\overline{w}, window, s')$
  let $s'.\overline{w'}.\mathtt{activedocument}.\mathtt{script} := script$
  stop $\langle\rangle$, s'
case $\langle \mathtt{SETSCRIPTSTATE}, window, scriptstate \rangle$
  let $\overline{w'} := \mathsf{GETWINDOW}(\overline{w}, window, s')$
  let $s'.\overline{w'}.\mathtt{activedocument}.\mathtt{scriptstate} := scriptstate$
  stop $\langle\rangle$, s'
case $\langle \mathtt{XMLHTTPREQUEST}, url, method, data, xhrreference \rangle$
  if $method \in \{\mathtt{CONNECT}, \mathtt{TRACE}, \mathtt{TRACK}\} \vee xhrreference \not\in V_{\text{process}} \cup \{\bot\}$ **then**
    stop
  if $url.\mathtt{host} \not\equiv docorigin.\mathtt{host} \vee url.\mathtt{protocol} \not\equiv docorigin.\mathtt{protocol}$ **then**
    stop
  if $method \in \{\mathtt{GET}, \mathtt{HEAD}\}$ **then**
    let $data := \langle\rangle$
    let $origin := \bot$
  else
    let $origin := docorigin$
  let $req := \langle \mathtt{HTTPReq}, \nu_4, method, url.\mathtt{host}, url.\mathtt{path}, url.\mathtt{parameters}, \langle\rangle, data \rangle$
  let $reference := \langle \mathtt{XHR}, s'.\overline{d}.\mathtt{nonce}, xhrreference \rangle$
  call $\mathsf{HTTP\_SEND}(reference,\ req,\ url,\ origin,\ referrer,\ referrerPolicy,\ a,\ s')$
case $\langle \mathtt{BACK}, window \rangle$
  let $\overline{w'} := \mathsf{GETNAVIGABLEWINDOW}(\overline{w},\ window,\ \bot,\ s')$
  call $\mathsf{NAVBACK}(\overline{w'},\ s')$
case $\langle \mathtt{FORWARD}, window \rangle$
  let $\overline{w'} := \mathsf{GETNAVIGABLEWINDOW}(\overline{w},\ window,\ \bot,\ s')$
  call $\mathsf{NAVFORWARD}(\overline{w'},\ s')$
case $\langle \mathtt{CLOSE}, window \rangle$
  let $\overline{w'} := \mathsf{GETNAVIGABLEWINDOW}(\overline{w},\ window,\ \bot,\ s')$
  remove $s'.\overline{w'}$ from the sequence containing it
  stop $\langle\rangle$, s'
case $\langle \mathtt{POSTMESSAGE}, window, message, origin \rangle$
  let $\overline{w'} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w'}.\mathtt{nonce} \equiv window$
  if $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{active} \equiv \top$
  $\hookrightarrow\ \wedge (origin \not\equiv \bot \implies s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{origin} \equiv origin)$ **then**
    let $s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{scriptinputs} := s'.\overline{w'}.\mathtt{documents}.\overline{j}.\mathtt{scriptinputs}$
    $\hookrightarrow\ +^{\langle\rangle} \langle \mathtt{POSTMESSAGE}, s'.\overline{w}.\mathtt{nonce}, docorigin, message \rangle$
  stop $\langle\rangle$, s'
case else
  stop

Algorithm 22 Web Browser Model: Process an HTTP response.

function ProcessResponse($response$, $reference$, $request$, $requestUrl$, key, a, f, s')
  if $\texttt{Set-Cookie} \in response.\texttt{headers}$ **then**
    for each $c \in^{\langle\rangle} response.\texttt{headers}\,[\texttt{Set-Cookie}]$, $c \in \mathsf{Cookies}$ **do**
      let $s'.\texttt{cookies}\,[request.\texttt{host}]$
      $\hookrightarrow$ $:= \mathsf{AddCookie}(s'.\texttt{cookies}\,[request.\texttt{host}]\,, c, requestUrl.\texttt{protocol})$
  if $\texttt{Strict-Transport-Security} \in response.\texttt{headers} \wedge requestUrl.\texttt{protocol} \equiv \texttt{S}$ **then**
    let $s'.\texttt{sts} := s'.\texttt{sts} +^{\langle\rangle} request.\texttt{host}$
  if $\texttt{Referer} \in request.\texttt{headers}$ **then**
    let $referrer := request.\texttt{headers}[\texttt{Referer}]$
  else
    let $referrer := \bot$
  if $\texttt{Location} \in response.\texttt{headers} \wedge response.\texttt{status} \in \{303, 307\}$ **then**
    let $url := response.\texttt{headers}\,[\texttt{Location}]$
    if $url.\texttt{fragment} \equiv \bot$ **then**
      let $url.\texttt{fragment} := requestUrl.\texttt{fragment}$
    let $method' := request.\texttt{method}$
    let $body' := request.\texttt{body}$
    if $\texttt{Origin} \in request.\texttt{headers}$
    $\hookrightarrow$ $\wedge\ request.\texttt{headers}[\texttt{Origin}] \not\equiv \diamond$
    $\hookrightarrow$ $\wedge\ (\langle url.\texttt{host}, url.\texttt{protocol}\rangle \equiv \langle request.\texttt{host}, requestUrl.\texttt{protocol}\rangle$
      $\hookrightarrow$ $\vee\ \langle request.\texttt{host}, requestUrl.\texttt{protocol}\rangle \equiv request.\texttt{headers}[\texttt{Origin}])$ **then**
      let $origin := request.\texttt{headers}[\texttt{Origin}]$
    else
      let $origin := \diamond$
    if $response.\texttt{status} \equiv 303 \wedge request.\texttt{method} \not\in \{\texttt{GET}, \texttt{HEAD}\}$ **then**
      let $method' := \texttt{GET}$
      let $body' := \langle\rangle$
    if $\exists\, \overline{w} \in \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{nonce} \equiv \pi_2(reference)$ **then** $\rightarrow$ Do not redirect XHRs.
      let $req := \langle \texttt{HTTPReq}, \nu_6, method', url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, body'\rangle$
      let $referrerPolicy := response.\texttt{headers}[\texttt{ReferrerPolicy}]$
      call HTTP_SEND($reference$, req, url, $origin$, $referrer$, $referrerPolicy$, a, s')
    else
      stop $\langle\rangle$, s'
  switch $\pi_1(reference)$ **do**
    case REQ
      let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{nonce} \equiv \pi_2(reference)$ **if possible;**
      $\hookrightarrow$ **otherwise stop** $\rightarrow$ normal response
      if $response.\texttt{body} \not\sim \langle *, * \rangle$ **then**
        stop $\langle\rangle$, s'
      let $script := \pi_1(response.\texttt{body})$
      let $scriptstate := \pi_2(response.\texttt{body})$
      let $d := \langle \nu_7, requestUrl, response.\texttt{headers}, referrer, script, scriptstate, \langle\rangle, \langle\rangle, \top\rangle$
      if $s'.\overline{w}.\texttt{documents} \equiv \langle\rangle$ **then**
        let $s'.\overline{w}.\texttt{documents} := \langle d \rangle$
      else
        let $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} \equiv \top$
        let $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} := \bot$
        remove $s'.\overline{w}.\texttt{documents}.(\overline{i} + 1)$ and all following documents
        $\hookrightarrow$ from $s'.\overline{w}.\texttt{documents}$
        let $s'.\overline{w}.\texttt{documents} := s'.\overline{w}.\texttt{documents} +^{\langle\rangle} d$
      stop $\langle\rangle$, s'
    case XHR
      let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$, $\overline{d}$ **such that** $s'.\overline{d}.\texttt{nonce} \equiv \pi_2(reference)$
      $\hookrightarrow$ $\wedge\ s'.\overline{d} = s'.\overline{w}.\texttt{activedocument}$ **if possible; otherwise stop**
      $\rightarrow$ process XHR response
      let $headers := response.\texttt{headers} - \mathsf{Set\text{-}Cookie}$
      let $s'.\overline{d}.\texttt{scriptinputs} := s'.\overline{d}.\texttt{scriptinputs} +^{\langle\rangle}$
      $\langle \texttt{XMLHTTPREQUEST}, headers, response.\texttt{body}, \pi_3(reference)\rangle$
      stop $\langle\rangle$, s'

Definition 56. The pair $((\langle a, f, m\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 23 (or any of the functions called therein), when given $(\langle a, f, m\rangle, s)$ as input, terminates with **stop** M, s', i.e., with output M and s'.

Recall that $\langle a, f, m\rangle$ is an (input) event and s is a (browser) state, M is a sequence of (output) protoevents, and s' is a new (browser) state (potentially with placeholders for nonces).

H. Definition of Web Browsers

Finally, we define web browser atomic Dolev-Yao processes as follows:

Definition 57 (Web Browser atomic Dolev-Yao Process). A web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, s_0^p)$ for a set I^p of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state $s_0^p \in Z_{\text{webbrowser}}$.

I. Helper Functions

In order to simplify the description of scripts, we use several helper functions.

a) CHOOSEINPUT (Algorithm 24): The state of a document contains a term, say $\mathit{scriptinputs}$, which records the input this document has obtained so far (via XHRs and postMessages). If the script of the document is activated, it will typically need to pick one input message from $\mathit{scriptinputs}$ and record which input it has already processed. For this purpose, the function $\mathsf{CHOOSEINPUT}(s', \mathit{scriptinputs})$ is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the scriptstate s' and chooses a yet unhandled input message from $\mathit{scriptinputs}$. The index of this message is then saved in the scriptstate (which is returned to the script).

b) CHOOSEFIRSTINPUTPAT (Algorithm 25): Similar to the function $\mathsf{CHOOSEINPUT}$ above, we define the function $\mathsf{CHOOSEFIRSTINPUTPAT}$. This function takes the term $\mathit{scriptinputs}$, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in $\mathit{scriptinputs}$ that matches $\mathit{pattern}$ and returns it. This function is typically used in places, where a script only processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

c) PARENTWINDOW: To determine the nonce referencing the parent window in the browser, the function $\mathsf{PARENTWINDOW}(\mathit{tree}, \mathit{docnonce})$ is used. It takes the term tree, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce $\mathit{docnonce}$, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the window which directly contains in its subwindows the window of the document referenced by $\mathit{docnonce}$. If there is no such window (which is the case if the script runs in a document of a top-level window), $\mathsf{PARENTWINDOW}$ returns $\perp$.

d) PARENTDOCNONCE: The function $\mathsf{PARENTDOCNONCE}(\mathit{tree}, \mathit{docnonce})$ determines (similar to $\mathsf{PARENTWINDOW}$ above) the nonce referencing the active document in the parent window in the browser . It takes the term tree, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce $\mathit{docnonce}$, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the active document in the window which directly contains in its subwindows the window of the document referenced by $\mathit{docnonce}$. If there is no such window (which is the case if the script runs in a document of a top-level window) or no active document, $\mathsf{PARENTDOCNONCE}$ returns $\mathit{docnonce}$.

e) SUBWINDOWS: This function takes a term tree and a document nonce $\mathit{docnonce}$ as input just as the function above. If $\mathit{docnonce}$ is not a reference to a document contained in tree, then $\mathsf{SUBWINDOWS}(\mathit{tree}, \mathit{docnonce})$ returns $\langle\rangle$. Otherwise, let $\langle \mathit{docnonce}, \mathit{location}, \langle\rangle, \mathit{referrer}, \mathit{script}, \mathit{scriptstate}, \mathit{scriptinputs}, \mathit{subwindows}, \mathit{active}\rangle$ denote the subterm of tree corresponding to the document referred to by $\mathit{docnonce}$. Then, $\mathsf{SUBWINDOWS}(\mathit{tree}, \mathit{docnonce})$ returns $\mathit{subwindows}$.

f) AUXWINDOW: This function takes a term tree and a document nonce $\mathit{docnonce}$ as input as above. From all window terms in tree that have the window containing the document identified by $\mathit{docnonce}$ as their opener, it selects one non-deterministically and returns its nonce. If there is no such window, it returns the nonce of the window containing $\mathit{docnonce}$.

g) AUXDOCNONCE: Similar to $\mathsf{AUXWINDOW}$ above, the function $\mathsf{AUXDOCNONCE}$ takes a term tree and a document nonce $\mathit{docnonce}$ as input. From all window terms in tree that have the window containing the document identified by $\mathit{docnonce}$ as their opener, it selects one non-deterministically and returns its active document's nonce. If there is no such window or no active document, it returns $\mathit{docnonce}$.

h) OPENERWINDOW: This function takes a term tree and a document nonce $\mathit{docnonce}$ as input as above. It returns the window nonce of the opener window of the window that contains the document identified by $\mathit{docnonce}$. Recall that the nonce identifying the opener of each window is stored inside the window term. If no document with nonce $\mathit{docnonce}$ is found in the tree tree or the document with nonce $\mathit{docnonce}$ is not directly contained in a top-level window, $\diamond$ is returned.

i) GETWINDOW: This function takes a term tree and a document nonce $\mathit{docnonce}$ as input as above. It returns the nonce of the window containing $\mathit{docnonce}$.

Algorithm 23 Web Browser Model: Main Algorithm.

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

let $s' := s$
if $s.\mathtt{isCorrupted} \not\equiv \bot$ **then**
    let $s'.\mathtt{pendingRequests} := \langle m, s.\mathtt{pendingRequests}\rangle$ → Collect incoming messages
    let $m' \leftarrow d_V(s')$
    let $a' \leftarrow \mathsf{IPs}$
    stop $\langle\langle a', a, m'\rangle\rangle$, s'
if $m \equiv \mathtt{TRIGGER}$ **then** → A special trigger message.
    let $\mathit{switch} \leftarrow \{\mathtt{script}, \mathtt{urlbar}, \mathtt{reload}, \mathtt{forward}, \mathtt{back}\}$
    if $\mathit{switch} \equiv \mathtt{script}$ **then** → Run some script.
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
        ↪ **if possible; otherwise stop** → Pointer to some window.
        let $\overline{d} := \overline{w} +^{\langle\rangle} \mathtt{activedocument}$
        call $\mathsf{RUNSCRIPT}(\overline{w}, \overline{d}, a, s')$
    else if $\mathit{switch} \equiv \mathtt{urlbar}$ **then** → Create some new request.
        let $\mathit{newwindow} \leftarrow \{\top, \bot\}$
        if $\mathit{newwindow} \equiv \top$ **then** → Create a new window.
            let $\mathit{windownonce} := \nu_1$
            let $w' := \langle \mathit{windownonce}, \langle\rangle, \bot\rangle$
            let $s'.\mathtt{windows} := s'.\mathtt{windows} +^{\langle\rangle} w'$
        else → Use existing top-level window.
            let $\overline{\mathit{tlw}} \leftarrow \mathbb{N}$ **such that** $s'.\overline{\mathit{tlw}}.\mathtt{documents} \neq \langle\rangle$
            ↪ **if possible; otherwise stop** → Pointer to some top-level window.
            let $\mathit{windownonce} := s'.\overline{\mathit{tlw}}.\mathit{nonce}$
        let $\mathit{protocol} \leftarrow \{\mathtt{P}, \mathtt{S}\}$
        let $\mathit{host} \leftarrow \mathsf{Doms}$
        let $\mathit{path} \leftarrow \mathbb{S}$
        let $\mathit{fragment} \leftarrow \mathbb{S}$
        let $\mathit{parameters} \leftarrow [\mathbb{S} \times \mathbb{S}]$
        let $\mathit{url} := \langle \mathtt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment}\rangle$
        let $\mathit{req} := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, \mathit{host}, \mathit{path}, \mathit{parameters}, \langle\rangle, \langle\rangle\rangle$
        call $\mathsf{HTTP\_SEND}(\langle \mathtt{REQ}, \mathit{windownonce}\rangle, \mathit{req}, \mathit{url}, \bot, \bot, \bot, a, s')$
    else if $\mathit{switch} \equiv \mathtt{reload}$ **then** → Reload some document.
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
        ↪ **if possible; otherwise stop** → Pointer to some window.
        let $\mathit{url} := s'.\overline{w}.\mathtt{activedocument}.\mathtt{location}$
        let $\mathit{req} := \langle \mathtt{HTTPReq}, \nu_2, \mathtt{GET}, \mathit{url}.\mathtt{host}, \mathit{url}.\mathtt{path}, \mathit{url}.\mathtt{parameters}, \langle\rangle, \langle\rangle\rangle$
        let $\mathit{referrer} := s'.\overline{w}.\mathtt{activedocument}.\mathtt{referrer}$
        let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\mathtt{nonce}, s')$
        call $\mathsf{HTTP\_SEND}(\langle \mathtt{REQ}, s'.\overline{w}.\mathtt{nonce}\rangle, \mathit{req}, \mathit{url}, \bot, \mathit{referrer}, \bot, a, s')$
    else if $\mathit{switch} \equiv \mathtt{forward}$ **then**
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
        ↪ **if possible; otherwise stop** → Pointer to some window.
        call $\mathsf{NAVFORWARD}(\overline{w}, s')$
    else if $\mathit{switch} \equiv \mathtt{back}$ **then**
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\mathtt{documents} \neq \langle\rangle$
        ↪ **if possible; otherwise stop** → Pointer to some window.
        call $\mathsf{NAVBACK}(\overline{w}, s')$
else if $m \equiv \mathtt{FULLCORRUPT}$ **then** → Request to corrupt browser
    let $s'.\mathtt{isCorrupted} := \mathtt{FULLCORRUPT}$
    stop $\langle\rangle$, s'
else if $m \equiv \mathtt{CLOSECORRUPT}$ **then** → Close the browser
    let $s'.\mathtt{secrets} := \langle\rangle$
    let $s'.\mathtt{windows} := \langle\rangle$
    let $s'.\mathtt{pendingDNS} := \langle\rangle$
    let $s'.\mathtt{pendingRequests} := \langle\rangle$
    let $s'.\mathtt{sessionStorage} := \langle\rangle$
    let $s'.\mathtt{cookies} \subset^{\langle\rangle} \mathsf{Cookies}$ **such that**
    ↪ $(c \in^{\langle\rangle} s'.\mathtt{cookies}) \iff (c \in^{\langle\rangle} s.\mathtt{cookies} \wedge c.\mathtt{content}.\mathtt{session} \equiv \bot)$
    let $s'.\mathtt{isCorrupted} := \mathtt{CLOSECORRUPT}$
    stop $\langle\rangle$, s'

else if $\exists\, \langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$ **such that**
$\hookrightarrow\ \pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \mathtt{HTTPResp}$ **then** → Encrypted HTTP response
  let $m' := \mathsf{dec_s}(m, \mathit{key})$
  if $m'.\mathtt{nonce} \not\equiv \mathit{request}.\mathtt{nonce}$ **then**
    stop
  remove $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \mathit{key}, f\rangle$ **from** $s'.\mathtt{pendingRequests}$
  call PROCESSRESPONSE(m', $\mathit{reference}$, $\mathit{request}$, url, key, a, f, s')
else if $\pi_1(m) \equiv \mathtt{HTTPResp} \wedge \exists\, \langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f\rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$ **such that**
$\hookrightarrow\ m.\mathtt{nonce} \equiv \mathit{request}.\mathtt{nonce}$ **then** → Plain HTTP Response
  remove $\langle \mathit{reference}, \mathit{request}, \mathit{url}, \bot, f\rangle$ **from** $s'.\mathtt{pendingRequests}$
  call PROCESSRESPONSE(m, $\mathit{reference}$, $\mathit{request}$, url, key, a, f, s')
else if $m \in \mathsf{DNSResponses}$ **then** → Successful DNS response
  if $m.\mathtt{nonce} \notin s.\mathtt{pendingDNS} \vee m.\mathtt{result} \notin \mathsf{IPs}$
  $\hookrightarrow\ \vee\ m.\mathtt{domain} \not\equiv s.\mathtt{pendingDNS}[m.\mathtt{nonce}].\mathtt{request}.\mathtt{host}$ **then**
    stop
  let $\langle \mathit{reference}, \mathit{message}, \mathit{url}\rangle := s.\mathtt{pendingDNS}[m.\mathtt{nonce}]$
  if $\mathit{url}.\mathtt{protocol} \equiv \mathtt{S}$ **then**
    let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
    $\hookrightarrow\ +^{\langle\rangle}\ \langle \mathit{reference}, \mathit{message}, \mathit{url}, \nu_3, m.\mathtt{result}\rangle$
    let $\mathit{message} := \mathsf{enc_a}(\langle \mathit{message}, \nu_3\rangle, s'.\mathtt{keyMapping}\,[\mathit{message}.\mathtt{host}])$
  else
    let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests}$
    $\hookrightarrow\ +^{\langle\rangle}\ \langle \mathit{reference}, \mathit{message}, \mathit{url}, \bot, m.\mathtt{result}\rangle$
  let $s'.\mathtt{pendingDNS} := s'.\mathtt{pendingDNS} - m.\mathtt{nonce}$
  stop $\langle\langle m.\mathtt{result}, a, \mathit{message}\rangle\rangle$, s'
stop

Algorithm 24 Function to retrieve an unhandled input message for a script.

function CHOOSEINPUT(s', $\mathit{scriptinputs}$)
  let iid **such that** $\mathit{iid} \in \{1, \cdots, |\mathit{scriptinputs}|\} \wedge \mathit{iid} \notin^{\langle\rangle} s'.\mathtt{handledInputs}$ **if possible;**
  $\hookrightarrow$ **otherwise return** $(\bot, s')$
  let $\mathit{input} := \pi_{\mathit{iid}}(\mathit{scriptinputs})$
  let $s'.\mathtt{handledInputs} := s'.\mathtt{handledInputs} +^{\langle\rangle} \mathit{iid}$
  return (input, s')

j) GETORIGIN: To extract the origin of a document, the function GETORIGIN(tree, $\mathit{docnonce}$) is used. This function searches for the document with the identifier $\mathit{docnonce}$ in the (cleaned) tree tree of the browser's windows and documents. It returns the origin o of the document. If no document with nonce $\mathit{docnonce}$ is found in the tree tree, $\diamond$ is returned.

k) GETPARAMETERS: Works exactly as GETORIGIN, but returns the document's parameters instead.

J. DNS Servers

Definition 58. A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s^d_0\}, R^d, s^d_0)$. It has a finite set of addresses I^d and its initial (and only) state s^d_0 encodes a mapping from domain names to addresses of the form

$$s^d_0 = \langle\langle \mathtt{domain}_1, a_1\rangle, \langle \mathtt{domain}_2, a_2\rangle, \ldots\rangle\ .$$

DNS queries are answered according to this table (if the requested DNS name cannot be found in the table, the request is ignored).

The relation $R^d \subseteq (\mathcal{E} \times \{s^d_0\}) \times (2^{\mathcal{E}} \times \{s^d_0\})$ of d above is defined by Algorithm 26.

K. Web Systems

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Algorithm 25 Function to extract the first script input message matching a specific pattern.

function CHOOSEFIRSTINPUTPAT($\mathit{scriptinputs}$, $\mathit{pattern}$)
  let i **such that** $i = \min\{j : \pi_j(\mathit{scriptinputs}) \sim \mathit{pattern}\}$ **if possible; otherwise return** $\bot$
  return $\pi_i(\mathit{scriptinputs})$

Algorithm 26 Relation of a DNS server R^d.

Input: $\langle a, f, m\rangle, s$
let $domain, n$ **such that** $\langle \texttt{DNSResolve}, domain, n\rangle \equiv m$ **if possible; otherwise stop** $\langle\rangle, s$
if $domain \in s$ **then**
  let $addr := s[domain]$
  let $m' := \langle \texttt{DNSResolved}, domain, addr, n\rangle$
  stop $\langle\langle f, a, m'\rangle\rangle, s$
stop $\langle\rangle, s$

Definition 59. A *web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon, Web, and Net of honest, web attacker, and network attacker processes, respectively.

Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \neq p$, $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs).

Every $p \in \mathsf{Hon}$ is a DY process which models either a *web server*, a *web browser*, or a *DNS server*. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.

The second component, $\mathcal{S}$, is a finite set of scripts such that $R^{\text{att}} \in \mathcal{S}$. The third component, script, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by script every $s \in \mathcal{S}$ is assigned its string representation $\mathsf{script}(s)$.

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for every $a \in \bigcup_{p\in\mathcal{W}} I^p$.

A *run* of $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E^0.

L. Generic HTTPS Server Model

This base model can be used to ease modeling of HTTPS server atomic processes. It defines placeholder algorithms that can be superseded by more detailed algorithms to describe a concrete relation for an HTTPS server.

Definition 60 (Base state for an HTTPS server). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $tlskeys \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

We note that in concrete instantiations of the generic HTTPS server model, there is no need to extract information from these subterms or alter these subterms.

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic web server in Algorithms 27–31, and the main relation in Algorithm 32.

Algorithm 27 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

function $\mathsf{HTTPS\_SIMPLE\_SEND}(reference,\ message,\ a,\ s')$
  let $s'.\texttt{pendingDNS}[\nu_{n0}] := \langle reference, message\rangle$
  stop $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, \nu_{n0}\rangle\rangle\rangle,\ s'$

Algorithm 28 Generic HTTPS Server Model: Default HTTPS response handler.

function $\mathsf{PROCESS\_HTTPS\_RESPONSE}(m,\ reference,\ request,\ a,\ f,\ s')$
  stop

Algorithm 29 Generic HTTPS Server Model: Default trigger event handler.

function $\mathsf{PROCESS\_TRIGGER}(a, s')$
  stop

Algorithm 30 Generic HTTPS Server Model: Default HTTPS request handler.

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
  stop

Algorithm 31 Generic HTTPS Server Model: Default handler for other messages.

function PROCESS_OTHER(m, a, f, s')
  stop

Algorithm 32 Generic HTTPS Server Model: Main relation of a generic HTTPS server

Input: $\langle a, f, m\rangle, s$
let $s' := s$
if $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ **then**
  let $s'.\mathtt{corrupt} := \langle\langle a, f, m\rangle, s'.\mathtt{corrupt}\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
if $\exists\, m_{\text{dec}}, k, k', \mathit{inDomain}$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\mathtt{tlskeys}$ **then**
  let n, method, path, $\mathit{parameters}$, $\mathit{headers}$, body **such that**
  $\hookrightarrow$ $\langle \mathtt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
  $\hookrightarrow$ **if possible; otherwise stop**
  call PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s')
else if $m \in \mathsf{DNSResponses}$ **then** $\rightarrow$ Successful DNS response
  if $m.\mathtt{nonce} \not\in s.\mathtt{pendingDNS} \vee m.\mathtt{result} \not\in \mathsf{IPs}$
  $\hookrightarrow$ $\vee\, m.\mathtt{domain} \not\equiv s.\mathtt{pendingDNS}[m.\mathtt{nonce}].2.\mathtt{host}$ **then**
    stop
  let $\mathit{reference} := s.\mathtt{pendingDNS}[m.\mathtt{nonce}].1$
  let $\mathit{request} := s.\mathtt{pendingDNS}[m.\mathtt{nonce}].2$
  let $s'.\mathtt{pendingRequests} := s'.\mathtt{pendingRequests} +^{\langle\rangle} \langle \mathit{reference}, \mathit{request}, \nu_{n1}, m.\mathtt{result}\rangle$
  let $\mathit{message} := \mathsf{enc_a}(\langle \mathit{request}, \nu_{n1}\rangle, s'.\mathtt{keyMapping}\,[\mathit{request}.\mathtt{host}])$
  let $s'.\mathtt{pendingDNS} := s'.\mathtt{pendingDNS} - m.\mathtt{nonce}$
  stop $\langle\langle m.\mathtt{result}, a, \mathit{message}\rangle\rangle$, s'
else if $\exists\, \langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle \in^{\langle\rangle} s'.\mathtt{pendingRequests}$
$\hookrightarrow$ **such that** $\pi_1(\mathsf{dec_s}(m, \mathit{key})) \equiv \mathtt{HTTPResp}$ **then** $\rightarrow$ Encrypted HTTP response
  let $m' := \mathsf{dec_s}(m, \mathit{key})$
  if $m'.\mathtt{nonce} \not\equiv \mathit{request}.\mathtt{nonce}$ **then**
    stop
  remove $\langle \mathit{reference}, \mathit{request}, \mathit{key}, f\rangle$ **from** $s'.\mathtt{pendingRequests}$
  call PROCESS_HTTPS_RESPONSE(m', $\mathit{reference}$, $\mathit{request}$, a, f, s')
else if $m \equiv \mathtt{TRIGGER}$ **then** $\rightarrow$ Process was triggered
  call PROCESS_TRIGGER(a, s')
else
  call PROCESS_OTHER(m, a, f, s')
stop

M. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [11].

Let $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ be a web system. In the following, we write $s_x = (S_x, E_x)$ for the states of a web system.

Definition 61 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p *emits* e iff there is a processing step in ρ of the form

$$(S^i, E^i, N^i) \xrightarrow[p \rightarrow E]{} (S^{i+1}, E^{i+1}, N^{i+1})$$

for some $i \geq 0$ and a sequence of events E with $e \in^{\langle\rangle} E$. We also say that p *emits* m iff $e = \langle x, y, m\rangle$ for some addresses x, y.

Definition 62. We say that a term t *is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \rightarrow s_{i+1}$ of a run $\rho = (s_0, s_1, \ldots)$)* if t is derivable from t' with the knowledge available to P, i.e.,

$$t \in d_\emptyset(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))$$

Definition 63. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 64. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'.

Definition 65. We say that *a browser b accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm 22).

Definition 66. We say that an atomic DY process p *knows a term t* in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 67. Let $N \subseteq \mathcal{N}$, $t \in \mathcal{T}_N(X)$, and $k \in \mathcal{T}_N(X)$. We say that k *appears only as a public key* in t, if

1) If $t \in N \cup X$, then $t \neq k$
2) If $t = f(t_1, \ldots, t_n)$, for $f \in \Sigma$ and $t_i \in \mathcal{T}_{\mathcal{N}}(X)$ $(i \in \{1, \ldots, n\})$, then $f = \mathsf{pub}$ or for all t_i, k appears only as a public key in t_i.

Definition 68. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm 21) and the first component of the $\mathit{command}$ output of the script relation is either `HREF`, `IFRAME`, `FORM`, or `XMLHTTPREQUEST` such that the browser issues the request r in the same step as a result.

Definition 69. We say that *an instance of the generic HTTPS server s accepted* a message (as a response to some request) if the server decrypted the message (if it was an HTTPS message) and called the function PROCESS_HTTPS_RESPONSE, passing the message and the request (see Algorithm 32).

For a run $\rho = s_0, s_1, \ldots$ of any $\mathcal{WS}$, we state the following lemmas:

Lemma 1. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b

(I) emits an HTTPS request of the form

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

(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, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$, and

(III) u never leaks k',

then all of the following statements are true:

(1) There is no state of $\mathcal{WS}$ where any party except for u knows k', thus no one except for u can decrypt m to obtain req.

(2) If there is a processing step $s_j \to s_{j+1}$ where the browser b leaks k to $\mathcal{W} \setminus \{u, b\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ or the browser is fully 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 the browsers' keymapping s_0.`keyMapping` (in its initial state).

(4) If b accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and b is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ 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 processes b and u.

PROOF. **(1)** follows immediately from the pre-conditions.

The process u never leaks k', and initially, the private key k' appears only as a public key in all other process states. As the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$, the other processes can never derive k'.

Thus, even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, and k' cannot be known to any other party. Thus, nobody except for u can derive req from m.

(2) We assume that b leaks k to $\mathcal{W} \setminus \{u, b\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and b and that the browser is not fully corrupted in s_j, and lead this to a contradiction.

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 64ff. of Algorithm 23) that is not used anywhere else. Further, the key is stored in the browser's state in $\mathit{pendingRequests}$. The information from $\mathit{pendingRequests}$ is not extracted or used anywhere else (in particular it is

not accessible by scripts). If the browser becomes closecorrupted prior to s_j (and after s_i), the key cannot be used anymore (compare Lines 46ff. of Algorithm 23). Hence, b does not leak k to any other party in s_j (except for u and b). This proves (2).

(3) Per the definition of browsers (Algorithm 23), a host header is always contained in HTTP requests by browsers. From Line 70 of Algorithm 23 we can see that the encryption key for the request req was chosen using the host header of the message. It is chosen from the `keyMapping` in the browser's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by b as a response to m has to be encrypted with k. The nonce k is stored by the browser in the $pendingRequests$ state information. The browser only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the browser checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and b (which did not leak it either, as u did not leak it and b is honest, see (2)). The browser b cannot send responses. This proves (4).

Corollary 1. In the situation of Lemma 1, as long as u does not leak the symmetric key k to $\mathcal{W} \setminus \{u, b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{u, b\}$ (i.e., $\nexists s' = (S', E') \in \rho$: $k \in d_{N^p}(S'(p))$).

Lemma 2. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \texttt{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 37 in Algorithm 22) the script in that document from an HTTPS response that was created by p.

PROOF. The origin of the document d is set only once: In Line 37 of Algorithm 22. The values (domain and protocol) used there stem from the information about the request (say, req) that led to the loading of d. These values have been stored in $pendingRequests$ between the request and the response actions. The contents of $pendingRequests$ are indexed by freshly chosen nonces and can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request req was added to $pendingRequests$ in Line 69 (or Line 72 which we can exclude as we will see later) of Algorithm 23. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in $pendingRequests$. When receiving the response to req, it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say, n). Only then the protocol part of the origin of the newly created document becomes S. The domain part of the origin (in our case dom) is taken directly from the $pendingRequests$ and is thus guaranteed to be unaltered.

From Line 70 of Algorithm 23 we can see that the encryption key for the request req was actually chosen using the host header of the message which will finally be the value of the origin of the document d. Since b therefore selects the public key $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that req was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 1 we see that the symmetric encryption key for the response, k, is only known to b and the respective web server. The same holds for the nonce n that was chosen by the browser and included in the request. Thus, no other party than p can encrypt a response that is accepted by the browser b and which finally defines the script of the newly created document.

Lemma 3. If in a processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \texttt{S}\rangle$ where $S_i(b).\texttt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then

- that request was initiated by a script that b extracted (in Line 37 in Algorithm 22) from an HTTPS response that was created by p, or
- that request is a redirect to a response of a request that was initiated by such a script.

PROOF. The browser algorithms create HTTP requests with an origin header by calling the HTTP_SEND function (Algorithm 18), with the origin being the fourth input parameter. This function adds the origin header only if this input parameter is not $\perp$.

The browser calls the HTTP_SEND function with an origin that is not $\perp$ only in the following places:

- Line 51 of Algorithm 21
- Line 72 of Algorithm 21
- Line 27 of Algorithm 22 ■

In the first two cases, the request was initiated by a script. The Origin header of the request is defined by the origin of the script's document. With Lemma 2 we see that the content of the document, in particular the script, was indeed provided by p.

In the last case (Location header redirect), as the origin is not $\diamond$, the condition of Line 17 of Algorithm 22 must have been true and the origin value is set to the value of the origin header of the request. In particular, this implies that an origin header

does not change during redirects (unless set to $\diamond$; in this case, the value stays the same in the subsequent redirects). Thus, the original request must have been created by the first two cases shown above.

The following lemma is similar to Lemma 1, but is applied to the generic HTTPS server (instead of the web browser).

Lemma 4. If in the processing step $s_i \rightarrow s_{i+1}$ of a run ρ of $\mathcal{WS}$ an honest instance s of the generic HTTPS server model

(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, for all processes $p \in \mathcal{W} \setminus \{u\}$, the private key k' appears only as a public key in $S^0(p)$,

(III) u never leaks k',

(IV) the instance model defined on top of the HTTPS server does not read or write the $pendingRequests$ subterm of its state,

(V) the instance model defined on top of the HTTPS server does not emit messages in $\mathsf{HTTPSRequests}$,

(VI) the instance model defined on top of the HTTPS server does not change the values of the $keyMapping$ subterm of its state, and

(VII) when receiving HTTPS requests of the form $\mathsf{enc_a}(\langle req', k_2\rangle, \mathsf{pub}(k'))$, u uses the nonce of the HTTP request req' only as nonce values of HTTPS responses encrypted with the symmetric key k_2,

(VIII) when receiving HTTPS requests of the form $\mathsf{enc_a}(\langle req', k_2\rangle, \mathsf{pub}(k'))$, u uses the symmetric key k_2 only for symmetrically encrypting HTTP responses (and in particular, k_2 is not part of a payload of any messages sent out by u),

then all of the following statements are true:

(1) There is no state of $\mathcal{WS}$ where any party except for u knows k', thus no one except for u can decrypt m to obtain req.

(2) If there is a processing step $s_j \rightarrow s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \rightarrow s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ or the process s 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 $S^0(s).\mathtt{keyMapping}$ (i.e., in the initial state of s).

(4) If s accepts a response (say, m') to m in a processing step $s_j \rightarrow s_{j+1}$ and s is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ 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 processes s and u.

PROOF. **(1)** follows immediately from the pre-conditions. The proof is the same as for Lemma 1:

The process u never leaks k', and initially, the private key k' appears only as a public key in all other process states. As the equational theory does not allow the extraction of a private key x from a public key $\mathsf{pub}(x)$, the other processes can never derive k'.

Thus, even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, and k' cannot be known to any other party. Thus, nobody except for u can derive req from m.

(2) We assume that some process leaks k to $\mathcal{W} \setminus \{u, s\}$ in the processing step $s_j \rightarrow s_{j+1}$ without u prior leaking the key k to anyone except for u and s and that the process s is not corrupted in s_j, and lead this to a contradiction.

The process s is honest in s_i. s emits HTTPS requests like m only in Line 18 of Algorithm 32:

- The message emitted in Line 3 of Algorithm 27 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 32
- There is no other place in the generic HTTPS server model where messages are emitted and due to pre-condition (V), the application-specific model does not emit HTTPS requests. ∎

The value k, which is the placeholder ν_{n1} in Algorithm 32, is only stored in the $pendingRequests$ subterm of the state of s, i.e., in $S^{i+1}(s).\mathtt{pendingRequests}$. Other than that, s only accesses this value in Line 19 of Algorithm 32, where it is only used to decrypt the response in Line 20 (in particular, the key is not propagated to the application-specific model, and the key cannot be contained within the payload of an response due to (VIII)). We note that there is no other line in the model of the generic HTTPS server where this subterm is accessed and the application-specific model does not access this subterm due to pre-condition (IV). Hence, s does not leak k to any other party in s_j (except for u and s). This proves (2).

(3) From Line 16 of Algorithm 32 we can see that the encryption key for the message m was chosen using the host header of the request. It is chosen from the $\mathtt{keyMapping}$ subterm of the state of s, which is never changed during ρ by the HTTPS server and never changed by the application-specific model due to pre-condition (VI). This proves (3).

(4)

Response was encrypted with k. An HTTPS response m' that is accepted by s as a response to m has to be encrypted with k:

The decryption key is taken from the $pendingRequests$ subterm of its state in Line 19 of Algorithm 32, where s only stores fresh nonces as keys that are added to requests as symmetric keys (see also Lines 15 and 16). The nonces (symmetric keys) are not used twice, or for other purposes than sending one specific request.

Only s and u can create the response. As shown previously, only s and u can derive the symmetric key (as s is honest in s_j). Thus, m' must have been created by either s or u.

s **cannot have created the response.** We assume that s emitted the message m' and lead this to a contradiction.

The generic server algorithms of s (when being honest) emit messages only in two places: In Line 3 of Algorithm 27, where a DNS request is sent, and in Line 18 of Algorithm 32, where a message with a different structure than m' is created (as m' is accepted by the server, m' must be a symmetrically encrypted ciphertext).

Thus, the instance model of s must have created the response m'.

Due to Precondition (IV), the instance model of s cannot read the `pendingRequests` subterm of its state. The symmetric key is generated freshly by the generic server algorithm in Lines 15 and 16 of Algorithm 32 and stored only in `pendingRequests`.

As the generic algorithms do not call any of the handlers with a symmetric key stored in `pendingRequests`., it follows that the instance model derived the key from a message payload in the instantiation of one of the handlers. Let $\tilde{m}$ denote this message payload.

As the server instance model cannot derive the symmetric key without processing a message from which it can derive the symmetric key, and as the server algorithm only create the original request m as the only message with the symmetric key as a payload, it follows that u must have created $\tilde{m}$, as no other process can derive the symmetric key from m.

However, when receiving m, u will use the symmetric key only as an encryption key, and in particular, will not create a message where the symmetric key is a payload (Precondition (VIII)).

Thus, the symmetric key cannot be derived by the instance of the server model, which is a contradiction to the statement that the instance model of s must have created the response m'.