Formal Security Analysis of the OpenID Financial-grade API 2.0

TECHNICAL REPORT – WP 1(B)

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

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 [23] 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 [23] 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)* [19], *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 second part of a formal security analysis of FAPI 2.0 (Baseline) 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 and prove 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], together with some defense mechanisms added to the specifications later (as a result of this analysis) and the changes described in Section III.

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 formal model, such as deviations from the specifications (as agreed on with the FAPI Working Group). 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. After some definitions, we describe the formal security properties in Section VIII. These security properties are proven in Section IX. 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 report.

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

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 [23] 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 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 2 and 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 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 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 7 and 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 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 10 and 11). The client now checks whether the issuer value matches the authorization server it intended to use before continuing (Step 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 13). FAPI mandates several security mechanisms at the token endpoint (Step 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 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 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 4.

The client can now use the access token to request protected resources from a resource server (Step 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 17). There are a number of ways for the resource server to verify access tokens, most notably *OAuth 2.0 Token Introspection* [22] 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.

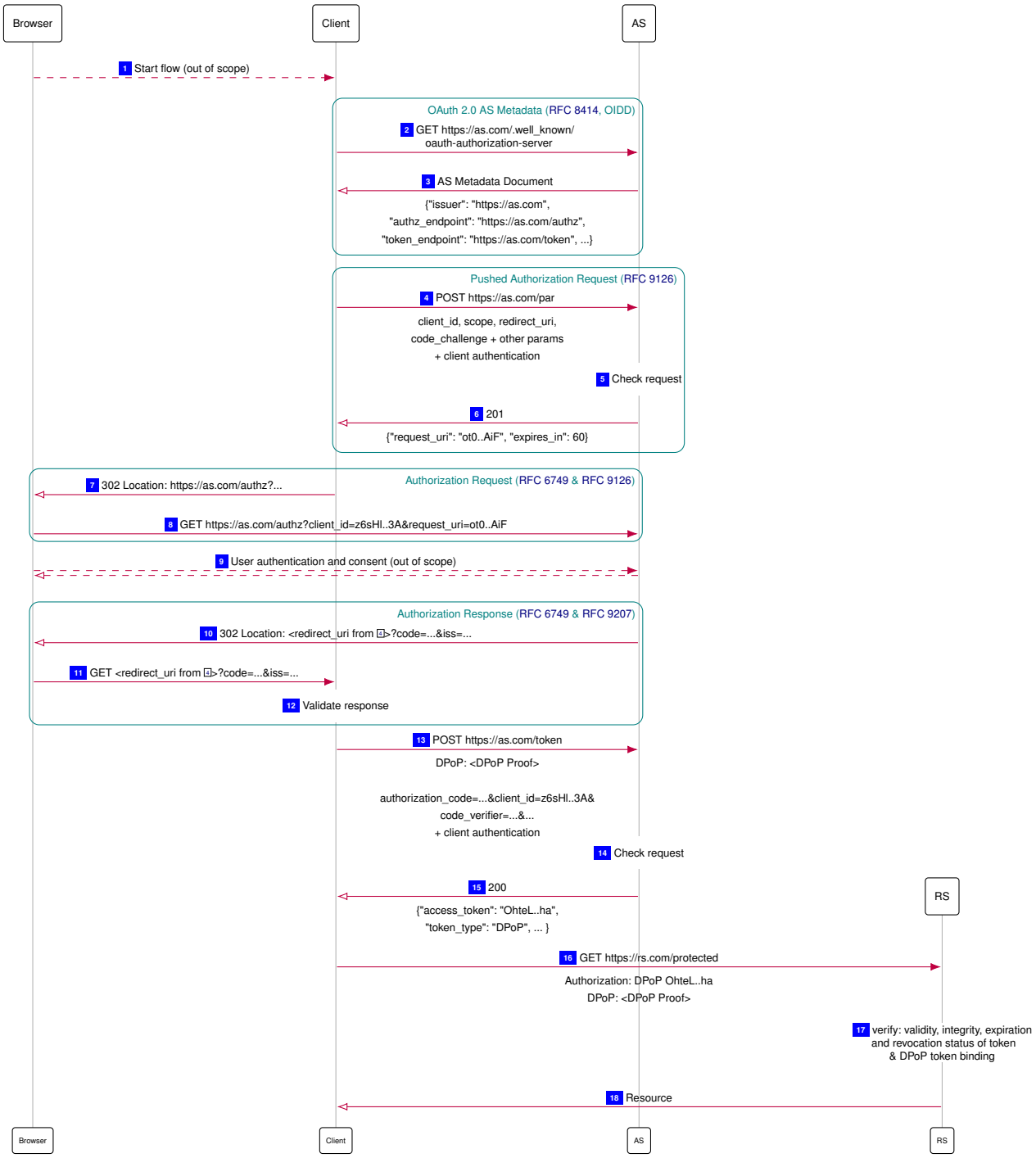

Figure 1. FAPI 2.0 Baseline Flow with DPoP token binding

III. Remarks on the Model

In this section, we explain some of the modeling decisions and assumptions of the formal model.

A. Defenses not mandated by FAPI

In order to be able to prove our security properties (see Section VIII), our model uses some security mechanisms which are either out of scope or not strictly mandated by FAPI 2.0. These are described and discussed here.

Preventing Cuckoo's Token Attack. A previous 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.

An attacker as described in FAPI 2.0 Attacker Model [5] is able to perform such an attack. However, the attack comes with a number of preconditions which the FAPI 2.0 Working Group expects to be met by very few real ecosystems. Hence, the FAPI 2.0 Working Group opted to – instead of making normative changes – add security considerations which explain the preconditions and recommend some mitigations and options to reduce the attack surface.

As the Cuckoo's Token Attack breaks any sensible authorization property,[3] including the one outlined in the FAPI 2.0 Attacker Model, we had to incorporate a fix for this attack into the formal model. In order to prevent "masking" other attacks, the FAPI 2.0 Working Group agreed on the following change to the protocol:

When a client receives an access token AT from an authorization server as, the client makes sure to only use AT to access resources which are managed by as. This prevents the Cuckoo's Token Attack, which relies on the client using an access token provided by an attacker-controlled authorization server to access resources which are managed by another (honest) authorization server. The formal model incorporates this change in Line 7 of Algorithm 4.

Preventing DPoP Proof Replay at the Resource Server. When DPoP [7] is used for access token sender constraining and an attacker gets hold of a resource request's plaintext, the attacker can replay the resource request to the resource server. The FAPI 2.0 Attacker Model states that FAPI 2.0 aims to be secure in the presence of such an attacker (A7 Attacker in [5]). However, the aforementioned replay attack breaks the desired authorization property as it enables an attacker to access protected resources.

The DPoP specification mentions this attack and suggests the use of the `jti` claim in the DPoP JWT to ensure DPoP proofs are used only once. This suggestion is not a requirement though: neither DPoP, nor FAPI 2.0 mandate resource servers to use this mechanism.

In practice, such replay attacks would require the attacker to eavesdrop on the resource request and place its replayed request in a relatively short time, because servers are required to only accept DPoP proofs for a very limited amount of time after their creation (see [7, Sec. 11.1]). Additionally, DPoP offers a second (also optional) countermeasure to such attacks: server-provided nonces, which the client has to include in its DPoP proofs.

In our formal model, we use the DPoP server-provided nonce mechanism for the communication between client and resource server to prevent the aforementioned DPoP replay attacks (see Line 8 of Algorithm 12, Line 38 of Algorithm 12, Line 56 of Algorithm 2, and Line 19 of Algorithm 4). However, we note that our formal model does not use server-provided nonces in the communication between client and authorization server.

We note that – contrary to the exact requirements of the DPoP specification, but agreed on with the FAPI Working Group – our resource server model treats DPoP nonces as "strictly one time use".

CSRF Protection at Client's /startLogin Endpoint. Usually, a FAPI 2.0 flow is initiated by some kind of interaction between the user's browser and a client, but this interaction is out of scope of the FAPI 2.0. Our model assumes that a user visits some client's index page which contains a form through which the user selects the authorization server she wants to use. This form is then submitted to the client's /startLogin endpoint – note that this is still out of scope of FAPI 2.0.

However, as our session integrity property, and in particular Definition 19, refers to the user submitting this form, we have to prevent an attacker from performing a CSRF attack in which the user's browser submits a similar form from the attacker's origin. As the model's browser always sends an Origin-Header, the client can just check that header for CSRF protection (see Line 6 of Algorithm 1). We once again emphasize that this part of the model is out of scope of FAPI 2.0 and we note that relying on the Origin-Header for CSRF protection is problematic in many real ecosystems, because browsers are not required to send this header.

Preventing Session Hijacking Attacks for Sessions Between Clients and Browsers. In order to be able to link a browser to a FAPI flow, the client sets a cookie in the browser once the browser submitted the authorization server selection form (see Line 30 of Algorithm 2). This cookie's value is subsequently used by the client to recognize the browser, and in particular to send the user's resources to this browser – this models access to the user's resources like, e.g., provided by Dropbox.

[3] By giving the attacker access to an honest user's resources at an honest resource server using an access token issued by an honest authorization server

If an attacker can perform a Session Hijacking Attack on this cookie, the attacker gets access to the user's resources or the user is accessing resources of the attacker (which is equally problematic, e.g., when the user uploads sensitive information to the attacker's account). A network attacker could perform such an attack if the user tries to access some unprotected client endpoint by intercepting the request and injecting a malicious response with a Set-Cookie header, using a value issued by the client for the attacker. There are several ways in which a client can protect its users against such attacks, e.g., by HSTS preloading or by using cookies with the secure attribute and the __Host prefix.

In our model, we use the latter option to prevent Session Hijacking Attacks (see Line 30 of Algorithm 2).

B. Changes Adopted by FAPI After July 1st

As stated previously, this analysis is based on [6], i.e., the state of FAPI 2.0 on July 1st, 2022. Being a work in progress, some security-relevant parts of FAPI 2.0 changed since then – some of them based on our feedback to the FAPI Working Group. As agreed upon with the FAPI Working Group, our model and properties include the following noteworthy changes related to security.

Removal of Attacker A3b. The FAPI 2.0 Attacker Model contained an attacker capability which allowed attackers to read authorization responses. In particular, this includes attacker access to (not yet redeemed) authorization codes of honest clients. Our analysis unveiled a browser-swapping attack in which an attacker gets access to an honest user's resources (at an honest resource server, through an honest client, with an access token issued by an honest authorization server). We explain this attack in more detail in Section VI.

While some (partly) fixes for this type of attack were discussed, the FAPI Working Group deemed them to not be implementable. Hence, Attacker A3b was dropped from the FAPI 2.0 Attacker Model, i.e., FAPI 2.0 no longer tries to protect against an attacker with access to authorization responses (see [24]).

Effective Removal of Attacker A5. The A5 attacker had the ability to force a client into using a token endpoint that is not one of the honest authorization server. Hence, such an attacker had access to, e.g., authorization codes, leading to possible attacks such as the ones described in the previous paragraph.

However, as described in Section VI, this attacker was defined to only apply to clients which receive their token endpoint via untrusted channels, e.g., by configuration. One explicitly mentioned example of a trusted channel is OAuth2.0 Authorization Server Metadata, which was recently clarified to be mandated for FAPI 2.0 clients. I.e., the A5 attacker is not relevant for a FAPI 2.0 client.

C. Important Modeling Decisions

In addition to the security measures discussed above, there are several notable modeling decisions which we discuss in the following.

Grant Types. FAPI 2.0 mandates authorization servers and clients to support the authorization code grant as defined in RFC 6749, while explicitly excluding the implicit grant, the resource owner credentials grant, and OpenID Connect's hybrid flow. This setting allows authorization servers and their clients to use grant types other than the authorization code grant, but does not require them to support any other type. While other grant types may of course introduce weaknesses, our analysis focuses on what FAPI 2.0 mandates. Consequently, our formal model of FAPI 2.0 only includes the authorization code grant.

DPoP at the Authorization Server. DPoP [7] is an essential part of FAPI 2.0 and contains several modes and optional parts as well. One of these options is the server-provided nonce mechanism for authorization and resource servers. We already discussed the need for server-provided nonces in the interaction between client and resource servers above and refer the reader to that description for a short outline of this mechanism. However, contrary to the resource server – client interaction, we did not model server-provided nonces in the interaction between authorization server and client. Adding these nonces would only make the modeled protocol more secure, while the DPoP specification does not mandate implementers to use this mechanism. Hence, we proved our security properties without relying on authorization server-provided nonces.

Another notable optional part of DPoP is the `dpop_jkt` authorization request parameter, which allows an authorization server to bind an issued authorization code to the client's proof-of-possession key. As this mechanism once again only adds security to the protocol, we chose not to include it in our formal model.

ID Token Validation. FAPI 2.0 clients requesting an ID token must verify such a token, following the OpenID Connect Core [23] standard. The required verification steps usually include verification of a JWS signature over the token contents. However, Section 3.1.3.7 of the OpenID Connect Core standard allows a client to skip the signature verification and instead verify the token's issuer via its TLS certificate if it received the ID token via TLS-secured back channel communication, i.e., directly from the authorization server (called *Identity Provider* in the OpenID Connect documents). Since FAPI 2.0 does not allow the implicit or hybrid flows and mandates the use of TLS on the authorization server's token endpoint, these conditions – back channel and TLS-secured – are always met.

Hence, our modeled client does not check signatures on ID tokens and instead verifies that the token response is a (TLS-secured) response to a request to the correct authorization server (see Line 47 of Algorithm 2).

Session Integrity Only if Authorization Request Does Not Leak. The FAPI 2.0 Attacker Model states that FAPI 2.0 aims to be secure even if the authorization request leaks to an attacker. However, if an attacker uses such a leaked authorization request to obtain an authorization code bound to the attacker's account, the attacker may be able to inject this authorization code into the honest user's session, e.g., via a CSRF attack, which in turn leads to the honest user accessing the attacker's resources. We note that PKCE [25] does not prevent this attack.

Such an attack violates the session integrity property outlined in the FAPI 2.0 Attacker Model. However, fixing this issue without fundamental changes to the protocol is not possible. Hence, the FAPI 2.0 Working Group decided to weaken the session integrity property by restricting it to flows in which the authorization request did not leak. Our formal security property in Section VIII is formulated accordingly. We note that all other security properties are still proven to hold, even if the authorization request leaked.

Resource Request Leak at Resource Server. The A7 attacker of the FAPI 2.0 Attacker Model allows the attacker to access resource requests (in plain text). This includes access to tokens and DPoP proofs sent in the resource request. If one assumes that such a leak occurs *before* the resource server processes the (honest) client's request, an attacker could just replay the whole resource request and get access to an honest user's resources, thus violating the authorization property.

Hence, we model the leakage of the resource request at the resource server (Line 69 of Algorithm 12 and Line 22 of Algorithm 13). See Section VI for details on the modeling.

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-I), the FAPI 2.0 authorization servers (Section IV-J), and the FAPI 2.0 resource servers (Section IV-K).

A. Identities

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 $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

B. 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}, $\mathsf{Passwords}$, and $\mathsf{RScredentials}$:

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

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).

C. Passwords

Definition 2. Let $\mathsf{ownerOfSecret}\colon \mathsf{Passwords} \to \mathsf{B}$ be a mapping 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).

D. Resources

We model the management of resources as follows: We assume that each resource is managed by at most one authorization server. We also assume that resources are identified by URLs at the resource server. Thus, when getting a request to such a resource URL, the resource server has to

1) identify the authorization server that is managing the resource, and
2) identify the identity for which the access token was issued.

If the access token is a structured JWT, the resource server retrieves the identity from the subject field. Otherwise, the identity is retrieved from the introspection response.

For identifying the authorization server, we first define the URL paths of resources managed by a resource server, and then define a mapping from these paths to authorization servers.

Definition 3. For each $rs \in \mathsf{RS}$, let $\mathsf{resourceURLPath}^{rs} \subseteq \mathbb{S}$ be a finite set of strings. These are the URL paths identifying the resources managed by the resource server.[4]

Definition 4. For each $rs \in \mathsf{RS}$, let $\mathsf{supportedAuthorizationServer}^{rs} \subseteq \mathsf{AS}$ be a finite set of authorization servers. These are the authorization servers supported by the resource server.

Definition 5. For each $rs \in \mathsf{RS}$, let $\mathsf{authorizationServerOfResource}^{rs}\colon \mathsf{resourceURLPath}^{rs} \to \mathsf{supportedAuthorizationServer}^{rs}$ be a mapping that assigns an authorization server to each resource URL path suffix of resources managed by the resource server.

If the access token is valid and the resource is managed by an authorization server supported by the resource server, the resource server model responds with a fresh nonce that it stores under the identity of the resource owner and the path under which it returned the resource. By using fresh nonces, the resource server does not return a nonce twice – even for requests for the same path and the same resource owner (identified via token introspection or the `sub` claim in the access token). Without this, the authorization property would need to exclude the case that the resource owner granted some malicious client access to a resource at some point.

E. 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-J. RS is the set of processes representing resource servers, described in Section IV-K. C is the set of processes representing clients, described in Section IV-I. Finally, B is the set of processes representing browsers, including their users. They are described in Appendix A-G.

F. 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 6. A *client information dictionary* is a dictionary of the form $[\texttt{client\_id}\colon clientId, \texttt{client\_type}\colon clientType, \texttt{mtls\_skey}\colon mtlsSkey, \texttt{jwt\_skey}\colon jwtSkey]$ where $clientId \in \mathcal{T}_{\mathcal{N}}$, $clientType \in \{\texttt{mTLS\_mTLS}, \texttt{mTLS\_DPoP}, \texttt{pkjwt\_mTLS}, \texttt{pkjwt\_DPoP}\}$, $mtlsSkey \in K_{\text{tls}} \cup \{\diamond\}$, and $jwtSkey \in K_{\text{sign}} \cup \{\diamond\}$. We further require $jwtSkey \not\equiv \diamond$ if and only if $clientType \in \{\texttt{pkjwt\_mTLS}, \texttt{pkjwt\_DPoP}, \texttt{mTLS\_DPoP}\}$, as well as $tlsSkey \not\equiv \diamond$ if and only if $clientType \in \{\texttt{mTLS\_mTLS}, \texttt{pkjwt\_mTLS}, \texttt{mTLS\_DPoP}\}$. Let ClientInfos be the set of all client information dictionaries.

Definition 7. 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 $\forall\, \mathsf{as} \in \mathsf{AS}, d_{as} \in \mathsf{dom}(\mathsf{as}), d_{client}, d'_{client} \in \mathsf{Doms}\colon \mathsf{clientInfo}(d_{as}, d_{client}).\texttt{client\_id} \equiv \mathsf{clientInfo}(d_{as}, d'_{client}).\texttt{client\_id} \Rightarrow d_{client} \equiv d'_{client}$, 2) TLS keys are assigned according to tlskey, formalized as $\mathsf{clientInfo}(d_{as}, d_{client}).\texttt{tlsSkey} \in \{\diamond, \mathsf{tlskey}(d_{client})\}$, and 3) signing keys are assigned according to signkey, formally expressed as $\mathsf{clientInfo}(d_{as}, d_{client}).\texttt{jwtSkey} \in \{\diamond, \mathsf{signkey}(\mathsf{dom}^{-1}(d_{client}))\}$. Note that while this definition requires the

[4] A resource is managed by the resource server if and only if $resourceID \in \mathsf{resourceURLPath}^{rs}$.

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 8. A client $c \in \mathsf{C}$ has the client identifier *clientId* at an authorization server $as \in \mathsf{AS}$ if $\exists\, d_{client} \in \mathsf{dom}(c), d_{as} \in \mathsf{dom}(as)$ such that $\mathsf{clientInfo}(d_{as}, d_{client}).\texttt{client\_id} \equiv clientId$.

Definition 9. 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 clientInfoAS by $\mathsf{as} \mapsto \langle\{\langle cli.\texttt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid \exists\ d_{client} \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(\mathsf{as})\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli\}\rangle$ where $\mathsf{as\_cli}(cli) := [\texttt{client\_type}\colon cli.\texttt{client\_type}] +^{\langle\rangle}$

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

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 10. 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 $c \mapsto \langle\{[d_{as}\colon [\texttt{client\_id}\colon cli.\texttt{client\_id}, \texttt{client\_type}\colon cli.\texttt{client\_type}]] \mid \exists\ d_{client} \in \mathsf{dom}(c), d_{as} \in \mathsf{Doms}\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli\}\rangle$. 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.

G. 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*[5] 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.[6]

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 [21, 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.

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

[6] 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.

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.

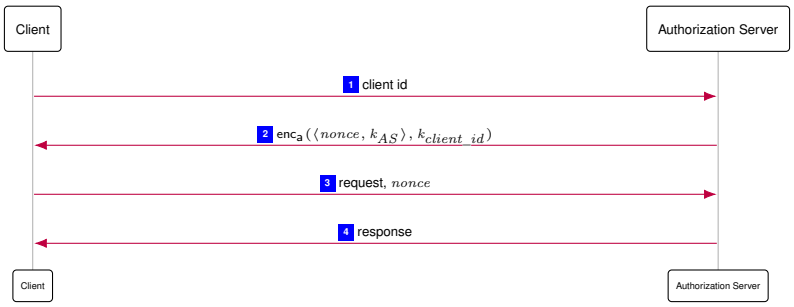

Figure 2. Overview of mTLS model

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.

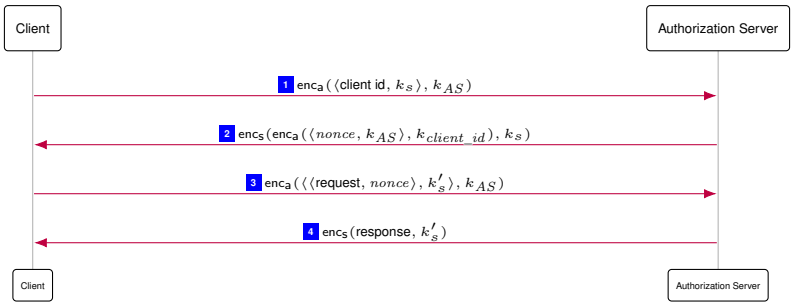

Figure 3. Detailed view on mTLS model

H. Additional HTTP Headers

In order to model DPoP, we extend the list of headers of Definition 40 with the following header: $\langle \texttt{DPoP}, p \rangle$ where $p \in \mathcal{T}_{\mathcal{N}}$ is (for honest senders) a DPoP proof (i.e., a signed JWT).

I. 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 11. A *state* $s \in Z^c$ *of a client* c is a term of the form $\langle DNSAddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, sessions, oauthConfigCache, jwksCache, asAccounts, mtlsCache, jwk, resourceASMapping, dpopNonces\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $corrupt \in \mathcal{T}_\mathcal{N}$, $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$, $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$ (all former components as in Definition 68), $sessions \in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $oauthConfigCache \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $jwksCache \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $asAccounts \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $mtlsCache \in \mathcal{T}_\mathcal{N}$, $jwk \in K_{\text{sign}}$, $resourceASMapping \in [\mathsf{Doms} \times [\mathbb{S} \times \mathsf{Doms}]]$, and $dpopNonces \in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$.

An *initial state* s_0^c *of* c is a state of c with

- $s_0^c.\texttt{DNSAddress} \equiv I^c$ (see Definition 67),
- $s_0^c.\texttt{pendingDNS} \equiv \langle\rangle$,
- $s_0^c.\texttt{pendingRequests} \equiv \langle\rangle$,
- $s_0^c.\texttt{corrupt} \equiv \bot$,
- $s_0^c.\texttt{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^c.\texttt{tlskeys} \equiv tlskeys^c$ (see Section IV-B),
- $s_0^c.\texttt{sessions} \equiv \langle\rangle$,
- $s_0^c.\texttt{oauthConfigCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{jwksCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{asAccounts} \equiv \mathsf{clientInfoClient}(c)$ (see Definition 10),
- $s_0^c.\texttt{mtlsCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{jwk} \equiv \mathsf{signkey}(c)$ (see Section IV-B),
- $s_0^c.\texttt{resourceASMapping}[domRS][resourceID] \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID))$, $\forall\, rs \in \mathsf{RS}$ and $\forall\, domRS \in \mathsf{dom}(rs)$ and $\forall\, resourceID \in \mathsf{resourceURLPath}^{rs}$ (a domain of the authorization server managing the resource stored at rs identified by $resourceID$), and
- $s_0^c.\texttt{dpopNonces} \equiv \langle\rangle$.

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 2–7. Note that in several places throughout these algorithms we use placeholders of the form ν_x to generate "fresh" nonces as described in our communication model (see Definition 25).

The script that is used by the client on its index page is described in Algorithm 8. In this script, to extract the current URL of a document, the function $\mathsf{GETURL}(tree, docnonce)$ is used. We define this function as follows: It searches for the document with the identifier $docnonce$ 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 $docnonce$ is found in the tree $tree$, $\diamond$ is returned.

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

function PROCESS_HTTPS_REQUEST(m, k, a, f, s') → **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 $m.\texttt{path} \equiv /$ **then** → Serve index page (start flow).
    let $m' := \texttt{enc}_\texttt{s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \mathit{headers}, \langle \texttt{script\_client\_index}, \langle\rangle\rangle\rangle, k)$ → Reply with *script_client_index*.
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if $m.\texttt{path} \equiv /\texttt{startLogin} \wedge m.\texttt{method} \equiv \texttt{POST}$ **then** → Start a new FAPI flow (probably from `script_client_index`)
    if $m.\texttt{headers}[\texttt{Origin}] \not\equiv \langle m.\texttt{host}, \texttt{S}\rangle$ **then**
      stop → Check the Origin header for CSRF protection to prevent attacker from starting a flow in the background (as this would trivially violate the session integrity property).
    let $\mathit{selectedAS} := m.\texttt{body}$
    let $\mathit{sessionId} := \nu_1$ → Session id is a freshly chosen nonce.
    let $s'.\texttt{sessions}[\mathit{sessionId}] := [\texttt{startRequest}: [\texttt{message}: m, \texttt{key}: k, \texttt{receiver}: a, \texttt{sender}: f], \texttt{selected\_AS}: \mathit{selectedAS}]$
    call PREPARE_AND_SEND_PAR($\mathit{sessionId}$, a, s') → Start an authorization flow with the AS (see Algorithm 6)
  else if $m.\texttt{path} \equiv /\texttt{redirect\_ep}$ **then** → User is being redirected after authentication to the AS.
    let $\mathit{sessionId} := m.\texttt{headers}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle]$
    if $\mathit{sessionId} \notin s'.\texttt{sessions}$ **then**
      stop
    let $\mathit{session} := s'.\texttt{sessions}[\mathit{sessionId}]$ → Retrieve session data.
    let $\mathit{selectedAS} := \mathit{session}[\texttt{selected\_AS}]$
    if $m.\texttt{parameters}[\texttt{iss}] \not\equiv \mathit{selectedAS}$ **then** → Check issuer parameter (RFC 9207).
      stop
    → Store browser's request for use in CHECK_ID_TOKEN (Algorithm 5) and PROCESS_HTTPS_RESPONSE (Algorithm 2)
    let $s'.\texttt{sessions}[\mathit{sessionId}][\texttt{redirectEpRequest}] := [\texttt{message}: m, \texttt{key}: k, \texttt{receiver}: a, \texttt{sender}: f]$
    call SEND_TOKEN_REQUEST($\mathit{sessionId}$, $m.\texttt{parameters}[\texttt{code}]$, a, s') → Retrieve a token from AS's token endpoint.
  stop → Unknown endpoint or malformed request.

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

function PROCESS_HTTPS_RESPONSE(m, *reference*, *request*, a, f, s')
  if *reference*[`responseTo`] $\equiv$ `MTLS` **then** → Client received an mTLS nonce (see Section IV-G)
    let m_{dec}, k' **such that** $m_{\text{dec}} \equiv \text{dec}_{\text{a}}(m.\texttt{body}, k') \wedge \langle dom, k' \rangle \in s'.\texttt{tlskeys}$ **if possible; otherwise stop**
    let *mtlsNonce*, *serverPubKey* **such that** $m_{\text{dec}} \equiv \langle mtlsNonce, serverPubKey \rangle$ **if possible; otherwise stop**
    if $serverPubKey \equiv s'.\texttt{keyMapping}[request.\texttt{host}]$ **then** → Verify sender of mTLS nonce
      let *clientId* := *reference*[`client_id`] → Note: If `client_id` $\notin$ *reference*, then *reference*[`client_id`] $\equiv \langle\rangle$
      let *pubKey* := *reference*[`pub_key`] → See note for client ID above
      let $s'.\texttt{mtlsCache} := s'.\texttt{mtlsCache} +^{\langle\rangle} \langle request.\texttt{host}, clientId, pubKey, mtlsNonce \rangle$
    stop $\langle\rangle, s'$
  let *sessionId* := *reference*[`session`]
  let *session* := $s'.\texttt{sessions}[sessionId]$
  let *selectedAS* := *session*[`selected_AS`]
  → Note: PREPARE_AND_SEND_PAR issues `CONFIG` and `JWKS` requests as required – once these get a response, we continue the PAR preparation by calling PREPARE_AND_SEND_PAR again.
  if *reference*[`responseTo`] $\equiv$ `CONFIG` **then**
    if $m.\texttt{body}[\texttt{issuer}] \not\equiv selectedAS$ **then** → Verify `issuer` value according to Sec. 3.3 of RFC 8414
      stop
    let $s'.\texttt{oauthConfigCache}[selectedAS] := m.\texttt{body}$
    call PREPARE_AND_SEND_PAR(*sessionId*, a, s')
  else if *reference*[`responseTo`] $\equiv$ `JWKS` **then**
    let $s'.\texttt{jwksCache}[selectedAS] := m.\texttt{body}$
    call PREPARE_AND_SEND_PAR(*sessionId*, a, s')
  else if *reference*[`responseTo`] $\equiv$ `PAR` **then**
    let *requestUri* := $m.\texttt{body}[\texttt{request\_uri}]$
    let $s'.\texttt{sessions}[sessionId][\texttt{request\_uri}] := requestUri$
    let *clientId* := *session*[`client_id`]
    let *request* := *session*[`startRequest`]
    → In the following, we construct the response to the initial request by some browser
    let *authEndpoint* := $s'.\texttt{oauthConfigCache}[selectedAS][\texttt{auth\_ep}]$
    → The authorization endpoint URL may include query components, which must be retained while also ensuring that no parameter appears more than once (Sec. 3.1 of RFC 6749). However, following Sec. 4 of RFC 9126 and Sec. 5 of RFC 9101 closely could introduce duplicates. We opted to overwrite `client_id` and `request_uri` parameters if present.
    let *authEndpoint*.`parameters`[`client_id`] := *clientId*
    let *authEndpoint*.`parameters`[`request_uri`] := *requestUri*
    let *headers* := [`Location`: *authEndpoint*]
    let *headers*[`Set-Cookie`] := [$\langle$`__Host-sessionId`$\rangle$: $\langle sessionId, \top, \top, \top \rangle$]
    let *response* := $\text{enc}_{\text{s}}(\langle \texttt{HTTPResp}, request[\texttt{message}].\texttt{nonce}, 303, headers, \langle\rangle \rangle, request[\texttt{key}])$
    let *leakAuthZReq* $\leftarrow \{\top, \bot\}$ → We assume that the authorization request, in particular `request_uri` and `client_id`, may leak to the attacker, see [5] and Section VI.
    if *leakAuthZReq* $\equiv \top$ **then**
      let *leak* := $\langle$`LEAK`, *authEndpoint*$\rangle$
      let *leakAddress* $\leftarrow$ IPs
      stop $\langle \langle request[\texttt{sender}], request[\texttt{receiver}], response \rangle, \langle leakAddress, request[\texttt{receiver}], leak \rangle \rangle, s'$
    else
      stop $\langle \langle request[\texttt{sender}], request[\texttt{receiver}], response \rangle \rangle, s'$
  else if *reference*[`responseTo`] $\equiv$ `TOKEN` **then**
    let *useAccessTokenNow* := $\top$
    if *session*[`scope`] $\equiv$ `openid` **then** → Non-deterministically decide whether to use the AT or check the ID token (if requested)
      let *useAccessTokenNow* $\leftarrow \{\top, \bot\}$
    if *useAccessTokenNow* $\equiv \top$ **then**
      call USE_ACCESS_TOKEN(*reference*[`session`], $m.\texttt{body}[\texttt{access\_token}]$, $request.\texttt{host}$, a, s')
    let *selectedAsTokenEp* := $s'.\texttt{oauthConfigCache}[selectedAS][\texttt{token\_ep}]$
    if $request.\texttt{host} \not\equiv selectedAsTokenEp.\texttt{host}$ **then**
      stop → Verify sender of HTTPS response is the expected authorization server (see OIDC Sec. 3.1.3.7)
    call CHECK_ID_TOKEN(*reference*[`session`], $m.\texttt{body}[\texttt{id\_token}]$, s')
  else if *reference*[`responseTo`] $\equiv$ `RESOURCE_USAGE` **then**
    → Reply to browser's request to the client's redirect endpoint (with the retrieved resource as payload)
    let *resource* := $m.\texttt{body}[\texttt{resource}]$
    let $s'.\texttt{sessions}[sessionId][\texttt{resource}] := resource$ → Store received resource
    let $s'.\texttt{sessions}[sessionId][\texttt{resourceServer}] := request.\texttt{host}$ → Store the domain of the RS
    let *request* := *session*[`redirectEpRequest`] → Data on browser's request to client's redirect endpoint
    let $m' := \text{enc}_{\text{s}}(\langle \texttt{HTTPResp}, request[\texttt{message}].\texttt{nonce}, 200, \langle\rangle, resource \rangle, request[\texttt{key}])$
    stop $\langle \langle request[\texttt{sender}], request[\texttt{receiver}], m' \rangle \rangle, s'$
  else if *reference*[`responseTo`] $\equiv$ `DPOP_NONCE` **then**
    let *dpopNonce* := $m.\texttt{body}[\texttt{nonce}]$
    let *rsDomain* := $request.\texttt{host}$
    let $s'.\texttt{dpopNonces}[rsDomain] := s'.\texttt{dpopNonces}[rsDomain] +^{\langle\rangle} dpopNonce$
    stop $\langle\rangle, s'$
  stop

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

function SEND_TOKEN_REQUEST($sessionId$, $code$, a, s')
  let $session := s'.\texttt{sessions}[sessionId]$
  let $pkceVerifier := session[\texttt{code\_verifier}]$
  let $selectedAS := session[\texttt{selected\_AS}]$
  let $headers := []$
  let $body := [\texttt{grant\_type}: \texttt{authorization\_code}, \texttt{code}: code, \texttt{redirect\_uri}: session[\texttt{redirect\_uri}]]$
  let $body[\texttt{code\_verifier}] := pkceVerifier$ → add PKCE Code Verifier (RFC 7636, Section 4.5)
  let $clientId := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_id}]$
  let $clientType := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_type}]$
  let $oauthConfig := s'.\texttt{oauthConfigCache}[selectedAS]$
  let $tokenEndpoint := oauthConfig[\texttt{token\_ep}]$
  → Client Authentication:
  if $clientType \in \{\texttt{mTLS\_mTLS}, \texttt{mTLS\_DPoP}\}$ **then** → mTLS client authentication
    let $body[\texttt{client\_id}] := clientId$ → RFC 8705 mandates `client_id` when using mTLS authentication
    let $mtlsNonce$ **such that** $\langle tokenEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle \in s'.\texttt{mtlsCache}$ **if possible; otherwise stop**
    let $authData := [\texttt{TLS\_AuthN}: mtlsNonce]$
    let $s'.\texttt{mtlsCache} := s'.\texttt{mtlsCache} -^{\langle\rangle} \langle tokenEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle$
  else if $clientType \in \{\texttt{pkjwt\_mTLS}, \texttt{pkjwt\_DPoP}\}$ **then** → `private_key_jwt` client authentication
    let $jwt := [\texttt{iss}: clientId, \texttt{sub}: clientId, \texttt{aud}: tokenEndpoint]$
    let $jws := \texttt{sig}(jwt, s'.\texttt{jwk})$
    let $authData := [\texttt{client\_assertion}: jws]$
  else
    stop → Invalid client type
  → Sender Constraining:
  if $clientType \equiv \texttt{mTLS\_mTLS}$ **then** → mTLS sender constraining (same nonce as for mTLS authN)
    let $mtlsNonce := authData[\texttt{TLS\_AuthN}]$
    let $body[\texttt{TLS\_binding}] := mtlsNonce$
  else if $clientType \equiv \texttt{pkjwt\_mTLS}$ **then** → mTLS sender constraining (fresh mTLS nonce)
    let $mtlsNonce$ **such that** $\langle tokenEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle \in s'.\texttt{mtlsCache}$ **if possible; otherwise stop**
    let $s'.\texttt{mtlsCache} := s'.\texttt{mtlsCache} -^{\langle\rangle} \langle tokenEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle$
    let $body[\texttt{TLS\_binding}] := mtlsNonce$
  else → Sender constraning using DPoP
    let $privKey := s'.\texttt{jwk}$ → get private key
    let $htu := tokenEndpoint$
    let $htu[\texttt{parameters}] := \langle\rangle$ → Section 4.2 of DPoP: without query
    let $htu[\texttt{fragment}] := \bot$ → Section 4.2 of DPoP: without fragment
    let $dpopJwt := [\texttt{headers}: [\texttt{jwk}: \texttt{pub}(privKey)]]$
    let $dpopJwt[\texttt{payload}] := [\texttt{htm}: \texttt{POST}, \texttt{htu}: htu]$
    let $dpopProof := \texttt{sig}(dpopJwt, privKey)$
    let $headers[\texttt{DPoP}] := dpopProof$ → add DPoP header; the $dpopJwt$ can be extracted with the extractmsg() function
  let $body := body +^{\langle\rangle} authData$
  let $message := \langle \texttt{HTTPReq}, \nu_2, \texttt{POST}, tokenEndpoint.\texttt{host}, tokenEndpoint.\texttt{path}, tokenEndpoint.\texttt{parameters}, headers, body\rangle$
  call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{TOKEN}, \texttt{session}: sessionId]$, $message$, a, s')

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

```
1: function USE_ACCESS_TOKEN(sessionId, token, tokenEPDomain, a, s')
2:     let session := s'.sessions[sessionId]
3:     let selectedAS := session[selected_AS]
4:     let rsDomain ← Doms    → This domain may or may not belong to a "real" resource server. If it belongs to the attacker, this request leaks the access token (but no mTLS nonce, nor a DPoP proof for an honest server).
       → Note: All paths except the mTLS and DPoP preparation endpoints are resource paths at the resource server.
5:     let resourceID ← 𝕊 such that resourceID ∉ {/mTLS-prepare, /DPoP-nonce}
6:     let url := ⟨URL, S, rsDomain, resourceID, ⟨⟩, ⊥⟩
7:     if s'.resourceASMapping[rsDomain][resourceID] ≢ tokenEPDomain then
8:         stop    → The AS from which the client received the AT is not managing the resource
       → The access token is sender-constraint, so the client must add a corresponding key proof.
9:     let clientType := s'.asAccounts[selectedAS][client_type]
10:    let clientId := s'.asAccounts[selectedAS][client_id]
11:    let body := []
12:    if clientType ∈ {mTLS_mTLS, pkjwt_mTLS} then    → mTLS sender constraining
13:        let mtlsNonce such that ⟨rsDomain, ⟨⟩, pubKey, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
14:        let body[TLS_binding] := mtlsNonce    → This nonce is not necessarily associated with the same of the client's keys as the access token. In such a case, the resource server will reject this request and the client has to try again.
15:        let headers := [Authorization: [Bearer: token]]    → FAPI 2.0 mandates to send access token in header
16:        let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨rsDomain, ⟨⟩, pubKey, mtlsNonce⟩
17:    else if clientType ∈ {mTLS_DPoP, pkjwt_DPoP} then    → DPoP sender constraining
18:        let privKey := s'.jwk    → get private key
19:        let dpopNonce such that dpopNonce ∈ s'.dpopNonces[rsDomain] if possible; otherwise stop
20:        let s'.dpopNonces[rsDomain] := s'.dpopNonces[rsDomain] −^⟨⟩ dpopNonce
21:        let htu := url
22:        let htu[parameters] := ⟨⟩    → Section 4.2 of DPoP: without query
23:        let htu[fragment] := ⊥    → Section 4.2 of DPoP: without fragment
24:        let dpopJwt := [headers: [jwk: pub(privKey)]]
25:        let dpopJwt[payload] := [htm: POST, htu: htu, ath: hash(token), nonce: dpopNonce]
26:        let dpopProof := sig(dpopJwt, privKey)
27:        let headers := [Authorization: [DPoP: token]]    → See Section 7.1 of DPoP
28:        let headers[DPoP] := dpopProof    → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
29:    let message := ⟨HTTPReq, ν3, POST, url.domain, url.path, ⟨⟩, headers, body⟩
30:    call HTTPS_SIMPLE_SEND([responseTo: RESOURCE_USAGE, session: sessionId], message, a, s')
```

Algorithm 5 Relation of a Client R^c – Check ID Token and log user in at c.

1: **function** CHECK_ID_TOKEN($sessionId$, $idToken$, s') → **Check ID Token validity and create service session.**
2: **let** $session := s'.\texttt{sessions}[sessionId]$ → Retrieve session data.
3: **let** $selectedAS := session[\texttt{selected\_AS}]$
4: **let** $oauthConfig := s'.\texttt{oauthConfigCache}[selectedAS]$ → Retrieve configuration for user-selected AS.
5: **let** $clientInfo := s'.\texttt{asAccounts}[selectedAS]$ → Retrieve client info used at that AS.
6: **let** $jwks := s'.\texttt{jwksCache}[selectedAS]$ → Retrieve signature verification key for AS.
7: **let** $data := \texttt{extractmsg}(idToken)$ → Extract contents of signed ID Token.
→ The following ID token checks are mandated by OIDC Sec. 3.1.3.7. Note that OIDC allows clients to skip ID token signature verification if the ID token is received *directly* from the authorization server (which it is here). Hence, we do not check the token's signature (see also Line 47 of Algorithm 2).
8: **if** $data[\texttt{iss}] \not\equiv selectedAS$ **then**
9: **stop** → Check the issuer; note that previous checks ensure $oauthConfig[\texttt{issuer}] \equiv selectedAS$
10: **if** $data[\texttt{aud}] \not\equiv clientInfo[\texttt{client\_id}]$ **then**
11: **stop** → Check the audience against own client id.
12: **if** $nonce \in session \wedge data[\texttt{nonce}] \not\equiv session[\texttt{nonce}]$ **then**
13: **stop** → If a nonce was used, check its value.
14: **let** $s'.\texttt{sessions}[sessionId][\texttt{loggedInAs}] := \langle selectedAS, data[\texttt{sub}]\rangle$ → User is now logged in. Store user identity and issuer of ID token.
15: **let** $s'.\texttt{sessions}[sessionId][\texttt{serviceSessionId}] := \nu_4$ → Choose a new service session id.
16: **let** $request := session[\texttt{redirectEpRequest}]$ → 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).
17: **let** $headers[\texttt{Set-Cookie}] := [\texttt{serviceSessionId}\colon \langle \nu_4, \top, \top, \top\rangle]$ → Create a cookie containing the service session id, effectively logging the user identified by $data[\texttt{sub}]$ in at this client.
18: **let** $m' := \texttt{enc}_\texttt{s}(\langle \texttt{HTTPResp}, request[\texttt{message}].\texttt{nonce}, 200, headers, \texttt{ok}\rangle, request[\texttt{key}])$
19: **stop** $\langle\langle request[\texttt{sender}], request[\texttt{receiver}], m'\rangle\rangle$, s'

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

function PREPARE_AND_SEND_PAR($sessionId$, a, s')
  let $redirectUris := \{\langle \texttt{URL}, \texttt{S}, d, /\texttt{redirect\_ep}, \langle\rangle, \bot\rangle \mid d \in \mathsf{dom}(c)\}$ → Set of redirect URIs for all domains of c.
  let $redirectUri \leftarrow redirectUris$ → Select a (potentially) different redirect URI for each authorization request
  let $session := s'.\texttt{sessions}[sessionId]$
  let $selectedAS := session[\texttt{selected\_AS}]$ → Authorization server selected by the user at the beginning of the flow.
  → Check whether the client needs to fetch authorization server metadata first and do so if required.
  if $selectedAS \notin s'.\texttt{oauthConfigCache}$ **then**
    let $path \leftarrow \{/.\texttt{well\_known}/\texttt{openid-configuration}, /.\texttt{well\_known}/\texttt{oauth-authorization-server}\}$
    let $message := \langle \texttt{HTTPReq}, \nu_5, \texttt{GET}, selectedAS, path, \langle\rangle, \langle\rangle, \langle\rangle\rangle$
    call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{CONFIG}, \texttt{session}: sessionId]$, $message$, a, s')
  let $oauthConfig := s'.\texttt{oauthConfigCache}[selectedAS]$
  → Check whether the client needs to fetch the authorization server's signature verification key first and do so if required.
  if $selectedAS \notin s'.\texttt{jwksCache}$ **then**
    let $url := oauthConfig[\texttt{jwks\_uri}]$
    let $message := \langle \texttt{HTTPReq}, \nu_5, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
    call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{JWKS}, \texttt{session}: sessionId]$, $message$, a, s')
  → Construct pushed authorization request
  let $parEndpoint := oauthConfig[\texttt{par\_ep}]$
  let $clientId := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_id}]$
  let $clientType := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_type}]$
  if $clientType \in \{\texttt{mTLS\_mTLS}, \texttt{mTLS\_DPoP}\}$ **then** → mTLS client authentication
    let $mtlsNonce$ **such that** $\langle parEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle \in s'.\texttt{mtlsCache}$ **if possible; otherwise stop**
    let $authData := [\texttt{TLS\_AuthN}: mtlsNonce]$
    let $s'.\texttt{mtlsCache} := s'.\texttt{mtlsCache} -^{\langle\rangle} \langle parEndpoint.\texttt{host}, clientId, \langle\rangle, mtlsNonce\rangle$
  else if $clientType \in \{\texttt{pkjwt\_mTLS}, \texttt{pkjwt\_DPoP}\}$ **then** → `private_key_jwt` client authentication
    let $jwt := [\texttt{iss}: clientId, \texttt{sub}: clientId, \texttt{aud}: parEndpoint]$
    let $jws := \mathsf{sig}(jwt, s'.\texttt{jwk})$
    let $authData := [\texttt{client\_assertion}: jws]$
  let $pkceVerifier := \nu_{\text{pkce}}$ → Fresh random value
  let $pkceChallenge := \mathsf{hash}(pkceVerifier)$
  let $parData := [\texttt{response\_type}: \texttt{code}, \texttt{code\_challenge\_method}: \texttt{S256}, \texttt{client\_id}: clientId,$
    ↪ $\texttt{redirect\_uri}: redirectUri, \texttt{code\_challenge}: pkceChallenge]$
  let $useOidc \leftarrow \{\top, \bot\}$ → Use of OIDC is optional
  if $useOidc \equiv \top$ **then**
    let $parData[\texttt{scope}] := \texttt{openid}$
  let $s'.\texttt{sessions}[sessionId] := s'.\texttt{sessions}[sessionId] +^{\langle\rangle} parData$
  let $parData := parData +^{\langle\rangle} authData$
  let $s'.\texttt{sessions}[sessionId][\texttt{code\_verifier}] := pkceVerifier$ → Store PKCE randomness in state
  let $authzReq := \langle \texttt{HTTPReq}, \nu_{\text{parNonce}}, \texttt{POST}, parEndpoint.\texttt{host}, parEndpoint.\texttt{path}, parEndpoint.\texttt{parameters}, \langle\rangle, parData\rangle$
  call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{PAR}, \texttt{session}: sessionId]$, $authzReq$, a, s')

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

function PROCESS_TRIGGER(a, s')
  let $action \leftarrow$ {`MTLS_PREPARE_AS`, `MTLS_PREPARE_RS`, `MTLS_PREPARE_MISCONFIGURED_TOKEN_EP`, `GET_DPOP_NONCE`}
  switch $action$ **do**
    case `MTLS_PREPARE_AS`
      let $server \leftarrow$ Doms **such that** $server \in s'$.`asAccounts` **if possible; otherwise stop**
      let $asAcc := s'$.`asAccounts`$[server]$
      let $clientId := asAcc$[`client_id`]
      let $body :=$ [`client_id`: $clientId$]
      let $message := \langle$`HTTPReq`, ν_{mtls}, `GET`, $server$, /`MTLS-prepare`, $\langle\rangle$, $\langle\rangle$, $body\rangle$
      call HTTPS_SIMPLE_SEND([`responseTo`: `MTLS`, `client_id`: $clientId$], $message$, a, s')
    case `MTLS_PREPARE_RS`
      let $resourceServer \leftarrow$ Doms → Note: This may or may not be a "real" resource server.
      let $domainAndKey \leftarrow s'$.`tlskeys`
      let $pubKey :=$ `pub`$(domainAndKey.2)$
      let $body :=$ [`pub_key`: $pubKey$]
      let $message := \langle$`HTTPReq`, ν_{mtls}, `GET`, $resourceServer$, /`MTLS-prepare`, $\langle\rangle$, $\langle\rangle$, $body\rangle$
      call HTTPS_SIMPLE_SEND([`responseTo`: `MTLS`, `pub_key`: $pubKey$], $message$, a, s')
    case `MTLS_PREPARE_MISCONFIGURED_TOKEN_EP`
      → This case allows the client to retrieve mTLS nonces from attacker-controlled servers and subsequently make requests to such servers. Without this case, the model would not capture attacks in which the client talks to attacker-controlled endpoints protected by mTLS.
      let $server \leftarrow$ Doms **such that** $server \in s'$.`asAccounts` **if possible; otherwise stop**
      let $asAcc := s'$.`asAccounts`$[server]$
      let $clientId := asAcc$[`client_id`]
      let $host \leftarrow$ Doms → Non-deterministically choose the domain instead of sending to the correct AS
      let $body :=$ [`client_id`: $clientId$]
      let $message := \langle$`HTTPReq`, ν_{mtls}, `GET`, $host$, /`MTLS-prepare`, $\langle\rangle$, $\langle\rangle$, $body\rangle$
      call HTTPS_SIMPLE_SEND([`responseTo`: `MTLS`, `client_id`: $clientId$], $message$, a, s')
    case `GET_DPOP_NONCE`
      → Our client uses DPoP server-provided nonces at the resource server. The resource server model offers a special endpoint to retrieve nonces.
      let $resourceServer \leftarrow$ Doms → Note: This may or may not be a "real" resource server.
      let $message := \langle$`HTTPReq`, ν_{DPoPreq}, `GET`, $resourceServer$, /`DPoP-nonce`, $\langle\rangle$, $\langle\rangle$, $\langle\rangle\rangle$
      call HTTPS_SIMPLE_SEND([`responseTo`: `DPOP_NONCE`], $message$, a, s')
  stop

Algorithm 8 Relation of $script\_client\_index$

Input: $\langle tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets\rangle$ → **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 $switch \leftarrow$ {`auth`, `link`} → Non-deterministically decide whether to start a login flow or to follow some link.
if $switch \equiv$ `auth` **then** → **Start login flow.**
  let $url :=$ GETURL$(tree, docnonce)$ → Retrieve URL of current document.
  let $id \leftarrow ids$ → Retrieve one of user's identities.
  let $as := id$.`domain` → Extract domain of AS from chosen id.
  let $url' :=$ ⟨`URL`, `S`, url.`host`, /`startLogin`, $\langle\rangle$, $\bot\rangle$ → Assemble request URL.
  let $command :=$ ⟨`FORM`, url', `POST`, as, $\bot\rangle$ → Post a form including the selected AS to the Client.
  stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$ → Finish script's run and instruct the browser to execute the command (i.e., to POST the form).
else → **Follow (random) link** to facilitate referrer-based attacks.
  let $protocol \leftarrow$ {`P`, `S`} → Non-deterministically select protocol (HTTP or HTTPS).
  let $host \leftarrow$ Doms → Non-det. select host.
  let $path \leftarrow \mathbb{S}$ → Non-det. select path.
  let $fragment \leftarrow \mathbb{S}$ → Non-det. select fragment part.
  let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$ → Non-det. select parameters.
  let $url :=$ ⟨`URL`, $protocol, host, path, parameters, fragment\rangle$ → Assemble request URL.
  let $command :=$ ⟨`HREF`, $url, \bot, \bot\rangle$ → Follow link to the selected URL.
  stop $\langle s, cookies, localStorage, sessionStorage, command\rangle$ → Finish script's run and instruct the browser to execute the command (follow link).

J. Authorization Servers

An authorization server $\mathsf{as} \in \mathsf{AS}$ 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 12. A *state* $s \in Z^{\mathsf{as}}$ *of an AS* as is a term of the form $\langle DNSAddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys,$ (signing key) $jwk, registrationRequests, clients, records, authorizationRequests, mtlsRequests, rsCredentials\rangle$ with: $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (as in Definition 68), $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$, $jwk \in K_{\text{sign}}$, $registrationRequests \in \mathcal{T}_{\mathcal{N}}$, $clients \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $records \in \mathcal{T}_{\mathcal{N}}$, $authorizationRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $mtlsRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, and $rsCredentials \in \mathcal{T}_{\mathcal{N}}$.

An *initial state* s_0^{as} *of* as is a state of as with $s_0^{\mathsf{as}}.\texttt{DNSaddress} \equiv I^{\mathsf{as}}$ (see Definition 67), $s_0^{\mathsf{as}}.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\texttt{corrupt} \equiv \bot$, $s_0^{\mathsf{as}}.\texttt{keyMapping}$ being the same as the keymapping for browsers, $s_0^{\mathsf{as}}.\texttt{tlskeys} \equiv tlskeys^{\mathsf{as}}$, $s_0^{\mathsf{as}}.\texttt{jwk} \equiv \mathsf{signkey}(\mathsf{as})$ (see Section IV-B), $s_0^{\mathsf{as}}.\texttt{registrationRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\texttt{clients} \equiv \mathsf{clientInfoAS}(\mathsf{as})$ (see Definition 9), $s_0^{\mathsf{as}}.\texttt{records} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\texttt{authorizationRequests} \equiv \langle\rangle$, $s_0^{\mathsf{as}}.\texttt{mtlsRequests} \equiv \langle\rangle$, and $s_0^{\mathsf{as}}.\texttt{rsCredentials} \equiv rsCreds$ where $rsCreds$ is a sequence and $\forall c\colon c \in rsCreds \Leftrightarrow (\exists d \in \mathsf{dom}(\mathsf{as}), rs \in \mathsf{Doms}\colon c \equiv \mathsf{secretOfRS}(d, rs))$.

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 10. Algorithm 11 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 $m.\texttt{path} \equiv /.\texttt{well-known/openid-configuration} \vee$
  $\hookrightarrow$ $m.\texttt{path} \equiv /.\texttt{well-known/oauth-authorization-server}$ **then** → We model both OIDD and RFC 8414.
    let $metaData := [\texttt{issuer}: m.\texttt{host}]$
    let $metaData[\texttt{auth\_ep}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{auth}, \langle\rangle, \bot \rangle$
    let $metaData[\texttt{token\_ep}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{token}, \langle\rangle, \bot \rangle$
    let $metaData[\texttt{par\_ep}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{par}, \langle\rangle, \bot \rangle$
    let $metaData[\texttt{introspec\_ep}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{introspect}, \langle\rangle, \bot \rangle$
    let $metaData[\texttt{jwks\_uri}] := \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{jwks}, \langle\rangle, \bot \rangle$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle\rangle, metaData \rangle, k)$
    stop $\langle \langle f, a, m' \rangle \rangle$, s'
  else if $m.\texttt{path} \equiv /\texttt{jwks}$ **then**
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle\rangle, \mathsf{pub}(s'.\texttt{jwk}) \rangle, k)$
    stop $\langle \langle f, a, m' \rangle \rangle$, s'
  else if $m.\texttt{path} \equiv /\texttt{auth}$ **then** → Authorization endpoint: Reply with login page.
    if $m.\texttt{method} \equiv \texttt{GET}$ **then**
      let $data := m.\texttt{parameters}$
    else if $m.\texttt{method} \equiv \texttt{POST}$ **then**
      let $data := m.\texttt{body}$
    let $requestUri := data[\texttt{request\_uri}]$
    if $requestUri \equiv \langle\rangle$ **then**
      stop → FAPI 2.0 mandates PAR, therefore a request URI is required
    let $authzRecord := s'.\texttt{authorizationRequests}[requestUri]$
    let $clientId := data[\texttt{client\_id}]$
    if $authzRecord[\texttt{client\_id}] \not\equiv clientId$ **then** → Check binding of request URI to client
      stop
    if $clientId \notin s'.\texttt{clients}$ **then**
      stop → Unknown client
    let $s'.\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}] := \nu_5$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle \langle \texttt{ReferrerPolicy}, \texttt{origin} \rangle \rangle, \langle \texttt{script\_as\_form}, [\texttt{auth2\_reference}: \nu_5] \rangle \rangle, k)$
    stop $\langle \langle f, a, m' \rangle \rangle$, s'
  else if $m.\texttt{path} \equiv /\texttt{auth2} \wedge m.\texttt{method} \equiv \texttt{POST} \wedge m.\texttt{headers}[\texttt{Origin}] \equiv \langle m.\texttt{host}, \texttt{S} \rangle$ **then** → Second step of authorization
    let $identity := m.\texttt{body}[\texttt{identity}]$
    let $password := m.\texttt{body}[\texttt{password}]$
    if $identity.\texttt{domain} \notin \mathsf{dom}(\texttt{as})$ **then**
      stop → This AS does not manage $identity$
    if $password \not\equiv \mathsf{secretOfID}(identity)$ **then**
      stop → Invalid user credentials
    let $auth2Reference := m.\texttt{body}[\texttt{auth2\_reference}]$
    let $requestUri$ **such that** $s'.\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}] \equiv auth2Reference$
    $\hookrightarrow$ **if possible; otherwise stop**
    let $authzRecord := s'.\texttt{authorizationRequests}[requestUri]$
    let $authzRecord[\texttt{subject}] := identity$
    let $authzRecord[\texttt{issuer}] := m.\texttt{host}$
    let $authzRecord[\texttt{code}] := \nu_1$ → Generate a fresh, random authorization code
    let $s'.\texttt{records} := s'.\texttt{records} +^{\langle\rangle} authzRecord$
    let $responseData := [\texttt{code}: authzRecord[\texttt{code}]]$
    if $authzRecord[\texttt{state}] \not\equiv \langle\rangle$ **then**
      let $responseData[\texttt{state}] := authzRecord[\texttt{state}]$
    let $redirectUri := authzRecord[\texttt{redirect\_uri}]$
    let $redirectUri.\texttt{parameters} := redirectUri.\texttt{parameters} \cup responseData$
    let $redirectUri.\texttt{parameters}[\texttt{iss}] := authzRecord[\texttt{issuer}]$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 303, \langle \langle \texttt{Location}, redirectUri \rangle \rangle, \langle\rangle \rangle, k)$
    stop $\langle \langle f, a, m' \rangle \rangle$, s'
  else if $m.\texttt{path} \equiv /\texttt{par} \wedge m.\texttt{method} \equiv \texttt{POST}$ **then** → Pushed Authorization Request
    if $m.\texttt{body}[\texttt{response\_type}] \not\equiv \texttt{code} \vee m.\texttt{body}[\texttt{code\_challenge\_method}] \not\equiv \texttt{S256}$ **then**
      stop
    let $authnResult := \mathsf{AUTHENTICATE\_CLIENT}(m, s')$ → Stops in case of errors/failed authentication
    let $clientId := authnResult.1$
    let $s' := authnResult.2$
    let $mtlsInfo := authnResult.3$
    if $clientId \not\equiv m.\texttt{body}[\texttt{client\_id}]$ **then**
      stop → Key used in client authentication is not registered for $m.\texttt{body}[\texttt{client\_id}]$
    let $redirectUri := m.\texttt{body}[\texttt{redirect\_uri}]$ → Clients are required to send redirect_uri with each request

```
63:         if redirectUri ≡ ⟨⟩ then
64:             stop
65:         if redirectUri.protocol ≢ S then
66:             stop
67:         let codeChallenge := m.body[code_challenge]    → PKCE challenge
68:         if codeChallenge ≡ ⟨⟩ then
69:             stop    → Missing PKCE challenge
70:         let requestUri := ν4    → Choose random URI
71:         let authzRecord := [client_id: clientId]
72:         let authzRecord[state] := m.body[state]
73:         let authzRecord[scope] := m.body[scope]
74:         if nonce ∈ m.body then
75:             let authzRecord[nonce] := m.body[nonce]
76:         let authzRecord[redirect_uri] := redirectUri
77:         let authzRecord[code_challenge] := codeChallenge
78:         let s′.authorizationRequests[requestUri] := authzRecord    → Store data linked to requestUri
79:         let m′ := encs(⟨HTTPResp, m.nonce, 201, ⟨⟩, [request_uri: requestUri]⟩, k)
80:         stop ⟨⟨f, a, m′⟩⟩, s′
81:     else if m.path ≡ /token ∧ m.method ≡ POST then
82:         if m.body[grant_type] ≢ authorization_code then
83:             stop
84:         let authnResult := AUTHENTICATE_CLIENT(m, s′)    → Stops in case of errors/failed authentication
85:         let clientId := authnResult.1
86:         let s′ := authnResult.2
87:         let mtlsInfo := authnResult.3
88:         let code := m.body[code]
89:         let codeVerifier := m.body[code_verifier]
90:         if code ≡ ⟨⟩ ∨ codeVerifier ≡ ⟨⟩ then
91:             stop    → Missing code or code_verifier
92:         let record, ptr such that record ≡ s′.records.ptr ∧ record[code] ≡ code ∧ code ≢ ⊥ ∧ ptr ∈ ℕ if possible; otherwise stop
93:         if record[client_id] ≢ clientId then
94:             stop
95:         if record[code_challenge] ≢ hash(codeVerifier) ∨ record[redirect_uri] ≢ m.body[redirect_uri] then
96:             stop    → PKCE verification failed or URI mismatch
97:         let clientType := s′.clients[clientId][client_type]
98:         if clientType ≡ pkjwt_DPoP ∨ clientType ≡ mTLS_DPoP then    → DPoP token binding
99:             let tokenType := DPoP
100:            let dpopProof := m.headers[DPoP]
101:            let dpopJwt := extractmsg(dpopProof)
102:            let verificationKey := dpopJwt[headers][jwk]
103:            if checksig(dpopProof, verificationKey) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
104:                stop    → Invalid DPoP signature (or empty jwk header)
105:            let dpopClaims := dpopJwt[payload]
106:            let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
107:            if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
108:                stop    → DPoP claims do not match corresponding message
109:            let cnfContent := [jkt: hash(verificationKey)]
110:        else if clientType ≡ pkjwt_mTLS ∨ clientType ≡ mTLS_mTLS then    → mTLS token binding
111:            let tokenType := Bearer
112:            let mtlsNonce := m.body[TLS_binding]
113:            if clientType ≡ mTLS_mTLS then    → Client used mTLS authentication, reuse data from authentication
114:                if mtlsNonce ≢ mtlsInfo.1 then
115:                    stop    → Client tried to use different mTLS key for authentication and token binding
116:            else    → Client did not use mTLS authentication
117:                let mtlsInfo such that mtlsInfo ∈ s′.mtlsRequests[clientId] ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
118:                let s′.mtlsRequests[clientId] := s′.mtlsRequests[clientId] −⟨⟩ mtlsInfo
119:            let mTlsKey := mtlsInfo.2    → mTLS public key of client
120:            let cnfContent := [x5t#S256: hash(mTlsKey)]
121:        else
122:            stop    → Client used neither DPoP nor mTLS
123:        let s′.records.ptr[code] := ⊥    → Invalidate code
124:        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: record[issuer]]
            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], sub: record[subject]]    → 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-G
        let clientId := m.body[client_id]
        let mtlsNonce := ν_3
        let clientKey := s'.clients[clientId][mtls_key]
        if clientKey ≡ ⟨⟩ ∨ clientKey ≡ pub(◇) then
            stop    → Client has no mTLS key
        let s'.mtlsRequests[clientId] := s'.mtlsRequests[clientId] +^⟨⟩ ⟨mtlsNonce, clientKey⟩
        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, s'.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') → Check client authentication in message m. Stops the current processing step in case of errors or failed authentication.
  if `client_assertion` $\in m.\texttt{body}$ **then** → `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**
    if $verificationKey \equiv \langle\rangle \vee verificationKey \equiv \mathsf{pub}(\diamond)$ **then**
      stop → Client has no jwt key
    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 → 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 $jwt[\texttt{aud}] \not\equiv \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{token}, \langle\rangle, \bot\rangle \wedge jwt[\texttt{aud}] \not\equiv m.\texttt{host}$ → `issuer` in AS metadata is just the host part $\hookrightarrow$ $\wedge\ jwt[\texttt{aud}] \not\equiv \langle \texttt{URL}, \texttt{S}, m.\texttt{host}, /\texttt{par}, \langle\rangle, \bot\rangle$ **then**
      stop → `aud` claim value is neither token, nor PAR endpoint nor AS issuer identifier
  else if `TLS_AuthN` $\in m.\texttt{body}$ **then** → mTLS client authentication
    let $clientId := m.\texttt{body}[\texttt{client\_id}]$ → 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 → Client authentication type mismatch
    let $s'.\texttt{mtlsRequests}[clientId] := s'.\texttt{mtlsRequests}[clientId] -^{\langle\rangle} mtlsInfo$
  else
    stop → 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 $\langle clientId, s', \bot\rangle$ → `private_key_jwt` client authentication, i.e., no mTLS info

Algorithm 11 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 $\langle s, cookies, localStorage, sessionStorage, command\rangle$

K. 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 13. A *state $s \in Z^{rs}$ of a resource server rs* is a term of the form $\langle DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, mtlsRequests$ (sequence of terms), $pendingResponses, resourceNonces$ (dict from ID to sequence of nonces), ids (sequence of ids), $asInfo, resourceASMapping, dpopNonces\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $pendingRequests \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $tlskeys \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (all former components as in Definition 68), $mtlsRequests \in \mathcal{T}_{\mathcal{N}}$, $pendingResponses \in \mathcal{T}_{\mathcal{N}}$, $resourceNonces \in [\mathsf{ID} \times \mathcal{T}_{\mathcal{N}}]$, $ids \subset \mathsf{ID}$, $asInfo \in [\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}]$, $resourceASMapping \in [\mathsf{resourceURLPath}^{rs} \times \mathcal{T}_{\mathcal{N}}]$, and $dpopNonces \in \mathcal{T}_{\mathcal{N}}$.

An *initial state s_0^{rs} of rs* is a state of rs with

- $s_0^{rs}.\mathtt{DNSaddress} \equiv I^{rs}$ (see Definition 67),
- $s_0^{rs}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{rs}.\mathtt{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^{rs}.\mathtt{tlskeys} \equiv tlskeys^{rs}$,
- $s_0^{rs}.\mathtt{mtlsRequests} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{pendingResponses} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{resourceNonces}$ being a dictionary where the resource server stores the resource nonces for each identity and resource id pair, initialized as $s_0^{rs}.\mathtt{resourceNonces}[id][resourceID] := \langle\rangle$, $\forall id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}, \forall resourceID \in \mathsf{resourceURLPath}^{rs}$
- $s_0^{rs}.\mathtt{ids} \subset^{\langle\rangle} \langle\mathsf{ID}\rangle$ such that $\forall id \in s_0^{rs}.\mathtt{ids} : \mathsf{governor}(id) \in \mathsf{supportedAuthorizationServer}^{rs}$, i.e., the resource server manages only resources of identities that are governed by one of the authorization server supported by the resource server,
- and for each domain of a supported authorization server $dom_{as} \in \mathsf{supportedAuthorizationSeverDoms}^{rs}$, let $s_0^{rs}.\mathtt{asInfo}$ contain a dictionary entry with the following values:
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{as\_introspect\_ep}] \equiv \langle \mathtt{URL}, \mathtt{S}, dom_{as}, /\mathtt{introspect}, \langle\rangle, \bot\rangle$ (the URL of the introspection endpoint of the authorization server)
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{as\_key}] \equiv \mathsf{signkey}(\mathsf{dom}^{-1}(dom_{as}))$ being the verification key for the authorization server
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{rs\_credentials}]$ being a sequence s.t. $\forall c\colon c \in^{\langle\rangle} s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{rs\_credentials}] \Leftrightarrow (\exists rsDom \in \mathsf{dom}(rs)\colon c \equiv \mathsf{secretOfRS}(dom_{as}, rsDom))$, i.e., the secrets used by the resource server for authenticating at the authorization server.
- $s_0^{rs}.\mathtt{resourceASMapping}[resourceID] \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID))$, $\forall resourceID \in \mathsf{resourceURLPath}^{rs}$ (a domain of the authorization server managing the resource identified by $resourceID$),
- $s_0^{rs}.\mathtt{dpopNonces} \equiv \langle\rangle$

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 12 and Algorithm 13.

Algorithm 12 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, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'
    else if m.path ≡ /DPoP-nonce then
        let freshDpopNonce := ν_dpop
        let s'.dpopNonces := s'.dpopNonces +^⟨⟩ freshDpopNonce
        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [nonce: freshDpopNonce]⟩, k)
        stop ⟨⟨f, a, m'⟩⟩, s'
    else
        let resourceID := m.path
        let responsibleAS := s'.resourceASMapping[resourceID]
        if responsibleAS ≡ ⟨⟩ then
            stop    → Resource is not managed by any of the supported ASs
        let asInfo := s'.asInfo[responsibleAS]
        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
                if dpopClaims[nonce] ∉ s'.dpopNonces then
                    stop    → Invalid DPoP nonce (see also Section III)
                if dpopClaims[ath] ≢ hash(accessToken) then
                    stop    → Invalid access token hash
                let s'.dpopNonces := s'.dpopNonces −^⟨⟩ dpopClaims[nonce]
                let cnfValue := [jkt: hash(verificationKey)]
            else
                stop    → Wrong Authorization header value
            let resource := ν_4    → Generate a fresh resource nonce
            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 s'.pendingResponses[requestId] := [expectedCNF: cnfValue, requestingClient: f,
                  ↪ resourceID: resourceID, originalRequest: m, originalRequestKey: k, resource: resource]
                let url := asInfo[as_introspect_ep]
                let rsCred ← asInfo[rs_credentials]    → 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, asInfo[as_key]) ≢ ⊤ then
                    stop    → Verification of AT signature failed
            let id := accessTokenContent[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 $s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] +^{\langle\rangle} resource$
let $m' := \texttt{enc}_\texttt{s}(\langle \texttt{HTTPResp}, m.\texttt{nonce}, 200, \langle\rangle, [\texttt{resource}{:}resource]\rangle, k)$
→ Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
let $leakingMessage := \langle \texttt{HTTPReq}, \nu_{\text{RRleak}}, \texttt{POST}, m.\texttt{domain}, m.\texttt{path}, m.\texttt{parameters}, m.\texttt{headers}, []\rangle$
let $leakAddress \leftarrow \mathsf{IPs}$
stop $\langle\langle f, a, m'\rangle, \langle leakAddress, a, \langle \texttt{LEAK}, leakingMessage\rangle\rangle\rangle, s'$
else
stop → Expected AT in Authorization header as mandated by FAPI

Algorithm 13 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 $origReq := pendingRequestInfo[\texttt{originalRequest}]$
let $originalRequestKey := pendingRequestInfo[\texttt{originalRequestKey}]$
let $resourceID := pendingRequestInfo[\texttt{resourceID}]$
let $resource := pendingRequestInfo[\texttt{resource}]$
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 $id :=$ m.body[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 $s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] +^{\langle\rangle} resource$
let $m' := \texttt{enc}_\texttt{s}(\langle \texttt{HTTPResp}, origReq.\texttt{nonce}, 200, \langle\rangle, [\texttt{resource}{:}resource]\rangle, originalRequestKey)$
→ Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
let $leakingMessage := \langle \texttt{HTTPReq}, \nu_{\text{RRleak}}, \texttt{POST}, origReq.\texttt{domain}, origReq.\texttt{path}, origReq.\texttt{parameters}, origReq.\texttt{headers}, []\rangle$
let $leakAddress \leftarrow \mathsf{IPs}$
stop $\langle\langle f, a, m'\rangle, \langle leakAddress, a, \langle \texttt{LEAK}, leakingMessage\rangle\rangle\rangle, s'$

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 laid out in that document are incorporated into the FAPI model. We note that FAPI 2.0 aims to be secure against any combination of the attacker types/capabilities described below.

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 34-36 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 version of FAPI 2.0 Attacker Model underlying this analysis [5] contains an attacker which has access to authorization responses. In particular, such an attacker has access to the authorization code contained in the authorization response. During our analysis, we discovered that such an attacker may perform a browser-swapping attack to get access to an honest user's resources.

Such an attack is depicted in Figure 4: The attacker poses as a regular user of an honest client and initiates a FAPI 2.0 flow (Step 1), selecting some honest authorization server to authenticate. The client continues this flow by retrieving the chosen authorization server's metadata document, followed by a pushed authorization request (Step 2). As usual, the client then answers (to the attacker's request) with a redirect to the authorization server (Step 3). Note that the flow is a regular FAPI 2.0 Baseline flow up to this point. Now, the attacker – instead of following the redirect – creates a clickable link to the redirection target URL and sends that link to the victim, e.g., disguised as some form of important notification by the authorization server (Step 5). The victim clicks this link (Step 6) and ends up at the authorization server's authorization endpoint, where the victim logs in and confirms (Step 7).[7] Following the victim's consent, the authorization server redirects the victim back to the client – this redirect contains the authorization response (Step 8). Following the FAPI 2.0 Attacker Model, this authorization response leaks to the attacker which may now send this exact authorization response to the client (Step 10) – just as if the attacker returned from the authorization server in an honest flow. After receiving the authorization response (from the attacker), the client retrieves an access token (and, optionally, an ID token) which is associated with the victim – and subsequently uses the victim's resources in a session with the attacker, i.e., giving the attacker access to the victim's resources.

Based on our feedback, the FAPI Working Group decided to drop the A3b attacker from the FAPI 2.0 Attacker Model. See also Section III-B.

[7]Depending on how the authorization server displays to the user what she is consenting to, this may require some social engineering by the attacker. However, we note that the client used in this attack is an honest client which the user/victim may trust.

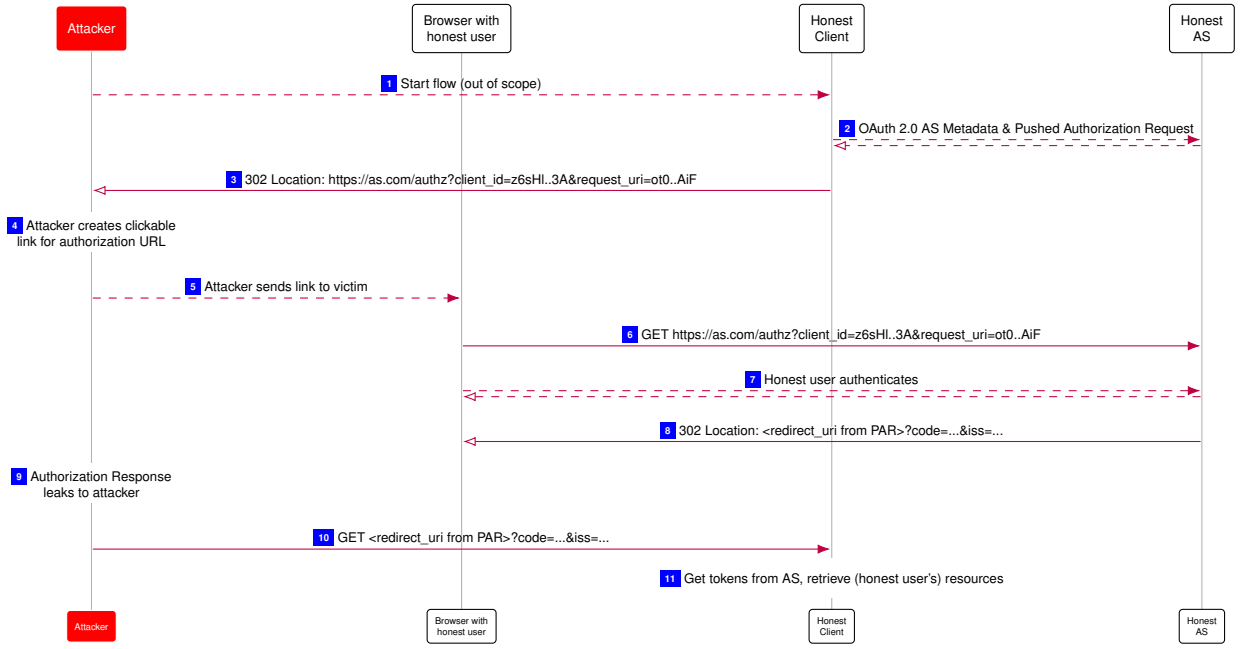

Figure 4. Browser Swapping attack based on Authorization Response leak

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. The FAPI 2.0 Attacker Model document states that this attacker is not relevant if the token endpoint URL has been received via a trusted source and explicitly mentions OAuth 2.0 Server Metadata. Using OAuth 2.0 Server Metadata used to be optional for clients (i.e., in the version of FAPI 2.0 underlying this analysis); however, this was changed and is now mandatory.

Hence, this attacker is not relevant and not included in our model.

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 after the server successfully checked the request and is sending the resource response, i.e., in Line 69 of Algorithm 12 and Line 22 of Algorithm 13: For access tokens that are sender-constrained using DPoP, the DPoP proof must not leak before the `nonce` value is invalidated. We note that within our model, we assume that the `nonce` value can be used exactly once at the resource server. We model the leakage of the application layer message, thus, the mTLS nonce is not leaked (see Line 67 of Algorithm 12 and Line 20 of Algorithm 13).

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."

Hence, we do not model this attacker.

VII. Auxiliary Definitions

The following definition captures that an access token was issued by an authorization server as, bound to a key k, and is associated with an identity id.

Definition 14 (Access Token bound to Key, Authorization Server and Identity). Let $k \in \mathcal{T}_{\mathcal{N}}$ be a term, $as \in \mathsf{AS}$ an authorization server, and $id \in \mathsf{ID}$ an identity. We say that a term t is an *access token bound to k, as, and id* in state S of the configuration (S, E, N) of a run ρ of a FAPI web system $\mathcal{FAPI}$, if there exists an entry $rec \in^{\langle\rangle} S(as).\texttt{records}$ such that

$$rec[\texttt{access\_token}] \equiv t \wedge \tag{1}$$
$$rec[\texttt{subject}] \equiv id \wedge \tag{2}$$
$$\big((rec[\texttt{cnf}] \equiv [\texttt{jkt}\colon \mathsf{hash}(k)]) \vee \tag{3}$$
$$(rec[\texttt{cnf}] \equiv [\texttt{x5t\#S256}\colon \mathsf{hash}(k)])\big) \tag{4}$$

(1) captures that the AS as created the access token.

(2) captures that the access token is associated with identity id (i.e., this identity authenticated previously at the authorization endpoint of the AS, and when the AT is redeemed at a RS, the RS will provide access to resources of this identity).

(3) and (4) capture that the access token is bound to a key. If (3) holds, then we say that the access token is bound via DPoP, otherwise, the token is bound via mTLS.

VIII. 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 all cases in which the resource is not directly accessed by the attacker, but also that no honest client provides the attacker access to such a resource.

Definition 15 (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 $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of $\mathcal{FAPI}$,
- every resource server $rs \in \mathsf{RS}$ that is honest in S^n,
- every identity $id \in^{\langle\rangle} s_0^{rs}.\texttt{ids}$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S^n,
- every processing step in ρ

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \to E^Q_{\text{out}}]{e^Q_{\text{in}} \to rs} (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$$

- every $resourceID \in \mathbb{S}$ with $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ being honest in S^Q,

it holds true that:

If $\exists x, y, k, m_{\text{resp}}.\langle x, y, \mathsf{enc_s}(m_{\text{resp}}, k)\rangle \in^{\langle\rangle} E^Q_{\text{out}}$ such that m_{resp} is an HTTP response with $m_{\text{resp}}.\texttt{body}[\texttt{resource}] \in^{\langle\rangle} S^{Q\prime}(rs).\texttt{resourceNonce}[id][resourceID]$, then

1) There exists a processing step

$$P = (S^P, E^P, N^P) \xrightarrow[rs \to E^P_{\text{out}}]{e^P_{\text{in}} \to rs} (S^{P\prime}, E^{P\prime}, N^{P\prime})$$

such that

a) either $P = Q$ or P prior to Q in ρ, and
b) e^P_{in} is an event $\langle x, y, \mathsf{enc_a}(\langle m_{\text{req}}, k_1\rangle, k_2)\rangle$ for some x, y, k_1, and k_2 where $m_{\text{req}} \in \mathcal{T}_{\mathcal{N}}$ is an HTTP request which contains a term (access token) t in its `Authorization` header, i.e., $t \equiv m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$, and

c) r was generated in P in Line 46 of Algorithm 12.

2) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, as, and id in S^Q (see Definition 14).
3) If there exists a client $c \in \mathsf{C}$ such that $k \equiv \mathsf{pub}(\mathsf{signkey}(c))$ or $k \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$ and $d_c \in \mathsf{dom}(c)$, and if c is honest in S^n, then r is not derivable from the attackers knowledge in S^n (i.e., $r \notin d_\emptyset(S^n(\mathsf{attacker}))$).

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 16 (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 17 (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 (in S) 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 18 (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 id in a login session identified by a nonce $lsid$ in processing step Q in ρ with

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

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

Definition 19 (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 8 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 $\langle \mathtt{Set\text{-}Cookie}, [\langle \mathtt{\_\_Host}, \mathtt{sessionId}\rangle\colon \langle lsid, \top, \top, \top\rangle]\rangle$. We then write $\mathsf{started}^Q_\rho(b, c, lsid)$.

Definition 20 (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 id for a login session identified by a nonce $lsid$ at the client c if there is a processing step $Q = (S, E, N) \to (S', E', N')$ in ρ 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 11, selected the identity id, and (2) we have that

- the $scriptstate$ of that document, when triggered in Q, contains a nonce $auth2Reference$ such that $scriptstate[\mathtt{auth2\_reference}] \equiv auth2Reference$, and
- there is a nonce $requestUri$ such that $S(as).\mathtt{authorizationRequests}[requestUri][\mathtt{auth2\_reference}] \equiv auth2Reference$, and
- $S(c).\mathtt{sessions}[lsid][\mathtt{request\_uri}] \equiv requestUri$.

We then write $\mathsf{authenticated}^Q_\rho(b, c, id, as, lsid)$.

Definition 21 (Resource Access). For a run ρ of a FAPI web system with a network attacker *$\mathcal{FAPI}$* we say that a browser $b \in \mathsf{B}$ gets access to a resource of identity u stored at resource server rs managed by authorization server as through the session of client c identified by the nonce $lsid$ in a processing step $Q = (S, E, N) \to (S', E', N')$ in ρ if c executes Line 55 of Algorithm 2 in Q, includes the resource r in the body of the HTTPS response that is sent out there, and it holds true that

1) $r \in^{\langle\rangle} S'(rs).\texttt{resourceNonces}[u][resourceId]$ and $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ (for some value $resourceId \in \mathcal{T}_{\mathcal{N}}$),
2) $\langle\langle \texttt{__Host}, \texttt{sessionid}\rangle, \langle lsid, y, z, z'\rangle\rangle \in^{\langle\rangle} S'(b).\texttt{cookies}[d]$ for $d \in \mathsf{dom}(c)$, $y, z, z' \in \mathcal{T}_{\mathcal{N}}$,
3) $S'(c).\texttt{sessions}[lsid][\texttt{resourceServer}] \in \mathsf{dom}(rs)$.
4) the request to which the client is responding contains a `Cookie` header with the cookie $\langle \texttt{__Host}, \texttt{sessionId}\rangle$ with the value $lsid$

We then write $\mathsf{accessesResource}^{Q}_{\rho}(b, r, u, c, rs, as, lsid)$.

Definition 22 (Client Leaked Authorization Request). Let $\mathcal{FAPI}$ be an FAPI web system with a network attacker. For a run ρ of $\mathcal{FAPI}$ with a processing step Q, a client $c \in \mathsf{C}$, a browser b, an authorization server $as \in \mathsf{AS}$, an identity id, a login session id $lsid$, and $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$, we say that c *leaked the authorization request for* $lsid$, if there is a processing step $Q' = (S, E, N) \xrightarrow[c \to E_{\text{out}}]{} (S', E', N')$ in ρ prior to Q such that in Q', c executes Line 36 of Algorithm 2 and there is a nonce $requestUri$ and an event $\langle x, y, m\rangle \in E_{\text{out}}$ with $m.1 \equiv \texttt{LEAK}$ and $m.2.\texttt{parameters}[\texttt{request\_uri}] \equiv requestUri$ such that $S'(c).\texttt{sessions}[lsid][\texttt{request\_uri}] \equiv requestUri$.

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 23 (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 $Q = (S, E, N) \to (S', E', N')$ in ρ, every browser b that is honest in S, every $as \in \mathsf{AS}$, every identity id, every client $c \in \mathsf{C}$ that is honest in S, every nonce $lsid$, and $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b, c, id, as, lsid)$.

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 the 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 24 (Session Integrity for Authorization). Let $\mathcal{FAPI}$ be a FAPI web system with a network attacker. 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 $Q = (S, E, N) \to (S', E', N')$ in ρ, 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 $\mathsf{accessesResource}^{\mathsf{Q}}_{\rho}(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), , then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b, c, u, as, lsid)$.

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

IX. Proofs

A. Helper Lemmas

Lemma 1 (Host of HTTP Request). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ and every process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ that is honest in S it holds true that if the generic HTTPS server calls $\mathsf{PROCESS\_HTTPS\_REQUEST}(m_{dec}, k, a, f, s)$ in Algorithm 31, then $m_{dec}.\texttt{host} \in \mathsf{dom}(p)$, for all values of k, a, f and s.

Proof. PROCESS_HTTPS_REQUEST is called only in Line 9 of Algorithm 31. The input message m is an asymetrically encrypted ciphertext. Intuitively, such a message is only decrypted if the process knows the private TLS key, where the private key used to decrypt is chosen (non-deterministically) according to the host of the decrypted message.

More formally, when PROCESS_HTTPS_REQUEST is called, the **stop** in Line 8 is not called. Therefore, it holds true that

$$\begin{aligned}
&\exists\, inDomain, k' : \langle inDomain, k'\rangle \in S(p).\texttt{tlskeys} \wedge m_{dec}.\texttt{host} \equiv inDomain \\
\Rightarrow\ &\exists\, inDomain, k' : \langle inDomain, k'\rangle \in \texttt{tlskeys}^{p} \wedge m_{dec}.\texttt{host} \equiv inDomain \\
\overset{\text{Def. (Section IV-B)}}{\Rightarrow}\ &\exists\, inDomain, k' : \langle inDomain, k'\rangle \in \{\langle d, tlskey(d)\rangle | d \in \mathsf{dom}(p)\} \wedge m_{dec}.\texttt{host} \equiv inDomain
\end{aligned}$$

From this, it follows directly that $m_{dec}.\texttt{host} \in \mathsf{dom}(p)$.

The first implication holds true due to $S(p).\texttt{tlskeys} \equiv s_0^p.\texttt{tlskeys} \equiv \texttt{tlskeys}^p$, as this sequence is never changed by any honest process $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ and due to the definitions of the initial states of clients, authorization servers, and resource servers (Definition 11, Definition 12, Definition 13). ■

Lemma 2 (Client's Signing Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every client $c \in \mathsf{C}$ that is honest in S, and every process p with $p \neq c$, all of the following hold true:

- $\mathsf{signkey}(c) \notin d_\emptyset(S(p))$
- $\mathsf{signkey}(c) \equiv s_0^c.\texttt{jwk}$
- $\mathsf{signkey}(c) \equiv S(c).\texttt{jwk}$

PROOF. $\mathsf{signkey}(c) \equiv s_0^c.\texttt{jwk}$ immediately follows from Definition 11. $\mathsf{signkey}(c) \equiv S(c).\texttt{jwk}$ follows from Definition 11 and by induction over the processing steps: state subterm `jwk` of a client is never changed.

The only places in which an honest client accesses the `jwk` state subterm are: Line 19 of Algorithm 3, Line 31 of Algorithm 3, Line 18 of Algorithm 4, and Line 24 of Algorithm 6.

In Line 19 of Algorithm 3 and Line 24 of Algorithm 6, the `jwk` state subterm is only used in a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key, i.e., cannot be extracted from the respective terms. Thus, it does not matter where these terms are stored or sent to. We conclude that these two usages of the `jwk` state subterm do not leak $\mathsf{signkey}(c)$ to any other process, in particular p.

In Line 31 of Algorithm 3 and Line 18 of Algorithm 4, the value of the `jwk` state subterm is stored in a variable $privKey$, which is then used in two places each:

1) In a $\mathsf{pub}(\cdot)$ term constructor (Line 35 of Algorithm 3 and Line 24 of Algorithm 4). The $privKey$ value cannot be extracted from these terms. Thus, it does not matter where these terms are stored or sent to.

2) In a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key (Line 37 of Algorithm 3 and Line 26 of Algorithm 4), i.e., cannot be extracted from the respective terms. Thus, it does not matter where these terms are stored or sent to.

We conclude that $\mathsf{signkey}(c) \notin d_\emptyset(S(p))$. ■

Lemma 3 (Code used in Token Request was received at Redirection Endpoint). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every processing step

$$P = (S, E, N) \xrightarrow[c \to E^Q_{\text{out}}]{e^Q_{\text{in}} \to c} (\tilde{S}, \tilde{E}, \tilde{N})$$

in ρ with $c \in \mathsf{C}$ being honest in S, it holds true that if Algorithm 2 (PROCESS_HTTPS_RESPONSE) is called in P with $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$, then there is a previous configuration (S', E', N') such that $request.\texttt{body}[\texttt{code}] \equiv S'(c).\texttt{sessions}[reference[\texttt{session}]][\texttt{redirectEpRequest}][\texttt{message}].\texttt{parameters}[\texttt{code}]$, with $request$ being an input parameter of PROCESS_HTTPS_RESPONSE.

PROOF. Let $sid := reference[\texttt{session}]$ be the session id with which PROCESS_HTTPS_RESPONSE is called in P.

Due to $reference[\texttt{responseTo}] \equiv \texttt{TOKEN}$, the corresponding request was sent in Line 41 of Algorithm 3 (SEND_TOKEN_REQUEST), as this is the only algorithm that uses this reference when sending a message. The code included in the request is the input parameter of SEND_TOKEN_REQUEST (see Line 6 of Algorithm 3).

SEND_TOKEN_REQUEST is called only in Line 21 of Algorithm 1, i.e., at the redirection endpoint (`/redirect_ep`) of the client, and the code is taken from the parameters of the redirection request. The redirection request is stored into $S'(c).\texttt{sessions}[sessionId][\texttt{redirectionEpRequest}]$ (with S' being the state of the input configuration in which the client processes the redirection request).

■

Lemma 4 (Authorization Server's Signing Key Does Not Leak). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration $Q = (S, E, N)$ in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every term t with $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(as))) \equiv \top$, and every process p with $p \neq as$, all of the following hold true:

- $\mathsf{signkey}(as) \notin d_\emptyset(S(p))$
- $\mathsf{signkey}(as) \equiv s_0^{as}.\texttt{jwk}$
- $\mathsf{signkey}(as) \equiv S(as).\texttt{jwk}$
- if t is *known* (Definition 74) to p in Q, then t was *created* (Definition 72) by as in a processing step s_e prior to Q in ρ

PROOF. $\mathsf{signkey}(as) \equiv s_0^{as}.\texttt{jwk}$ immediately follows from Definition 12. $\mathsf{signkey}(as) \equiv S(as).\texttt{jwk}$ follows from Definition 12 and by induction over the processing steps: state subterm `jwk` of a client is never changed.

By Definitions 11, 12, 13, 58, and Section IV-B, we have that no process (except as) initially knows $\mathsf{signkey}(as)$, i.e., $\mathsf{signkey}(as) \notin d_\emptyset(S^0(p))$.

The only places in which an honest authorization server accesses the `jwk` state subterm are: Line 11 of Algorithm 9, Line 127 of Algorithm 9, and Line 139 of Algorithm 9.

In Line 127 of Algorithm 9 and Line 139 of Algorithm 9, the `jwk` state subterm is only used in a $\mathsf{sig}(\cdot,\cdot)$ term constructor as signature key, i.e., cannot be extracted from the respective terms. Thus, it does not matter where these terms are stored or sent to. We conclude that these two usages of the `jwk` state subterm do not leak $\mathsf{signkey}(as)$ to any other process, in particular p.

In Line 11 of Algorithm 9, the `jwk` state subterm is only used in a $\mathsf{pub}(\cdot)$ term constructor, i.e., cannot be extracted from the constructed term. Thus, it does not matter where such a term is stored or sent to.

We conclude that $\mathsf{signkey}(as) \notin d_\emptyset(S(p))$.

To complete the proof, we now have to show that any term t with $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(as))) \equiv \top$ *known* to p in Q was *created* by as in a processing step s_e prior to Q in ρ:

By Definitions 11, 12, 13, 58, and Section IV-B, we have that no process (except as) initially knows such a term t, i.e., $t \notin d_\emptyset(S^0(p))$. Together with Definition 49 and Definition 72, this implies that t can only be *known* to p in some configuration Q' if t was contained in some event e "received" by p at an earlier point in ρ (i.e., e was the input event in a processing step in ρ with p). Since such an e is not part of E^0 (Definition 67), e must have been *emitted* by some process in a processing step s_e prior to Q' in ρ. Definition 49 and Definition 69 imply that p (or any other process $\neq as$) cannot have *emitted* e in s_e (i.e., cannot have created t in s_e).

Therefore, as must have *emitted* e and hence *created* t in s_e, i.e., prior to Q in ρ. ■

Lemma 5 (mTLS Nonce created by AS does not Leak). For every run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S^n, every client $c \in \mathsf{C}$ that is honest in S^n with client id $clientId$ at as, every $i \in \mathbb{N}$ with $0 \leq i \leq |S(as).\texttt{mtlsRequests}[clientId]|$, and every process p with $as \neq p \neq c$ it holds true that $mtlsNonce := S(as).\texttt{mtlsRequests}[clientId].i.1$ does not leak to p, i.e., $mtlsNonce \notin d_\emptyset(S^n(p))$.

PROOF. Initially, the `mtlsRequests` subterm of the authorization server's state is empty, i.e., $S^0(as).\texttt{mtlsRequests} \equiv \langle\rangle$ (Definition 12). An authorization server only adds values to the `mtlsRequests` subterm in Line 161 of Algorithm 9, where the mTLS nonce is chosen as a fresh nonce (Line 157 of Algorithm 9).

Let $(S^i, E^i, N^i) \to (S^{i'}, E^{i'}, N^{i'})$ be the processing step in which the nonce is chosen (note that (S^i, E^i, N^i) is prior to (S, E, N) in ρ). In the same processing step, the authorization server sends out the nonce in Line 163 of Algorithm 9, asymmetrically encrypted with the public key

$$
\begin{aligned}
& clientKey \\
\equiv\ & S^i(as).\texttt{clients}[clientId][\texttt{mtls\_key}] && \text{(Line 158, Algorithm 9)} \\
\equiv\ & s_0^{as}.\texttt{clients}[clientId][\texttt{mtls\_key}] && \text{(value is never changed)} \\
\equiv\ & \mathsf{clientInfoAS}(as)[clientId][\texttt{mtls\_key}] && \text{(Definition 12)} \\
\equiv\ & \langle\{\langle cli.\texttt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as)\}\rangle[clientId][\texttt{mtls\_key}] && \text{(Definition 9)}
\end{aligned}
$$

with $cli = \mathsf{clientInfo}(d_{as}, d_c)$ and $\mathsf{as\_cli}$ as in Definition 9. As $clientId$ is the identifier of c at as and as no two clients have the same identifier at an authorization server (see Definition 7), it follows that $d_c \in \mathsf{dom}(c)$.

As there is a dictionary entry in $S^i(as).\texttt{clients}[clientId]$ with the key `mtls_key` (Line 159, Algorithm 9), it follows that

$$
\begin{aligned}
& clientKey \\
\equiv\ & \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{tlsSKey}) && (d_{as} \in \mathsf{dom}(as),\ d_c \in \mathsf{dom}(c);\ \text{see also Definition 12}) \\
\equiv\ & \mathsf{pub}(\mathsf{tlskey}(d_c)) && (d_c \in \mathsf{dom}(c))
\end{aligned}
$$

The corresponding private key is $\mathsf{tlskey}(d_c) \in tlskeys^c$, which is only known to c (Lemma 2). (The $mtlsNonce$ saved in `mtlsRequests` is not sent in any other place).

This implies that the encrypted nonce can only be decrypted by c. c decrypts messages only in Line 3 of Algorithm 2. (The only other place where a message is decrypted asymmetrically by c is in the generic HTTPS server (Line 7 of Algorithm 31), where the process would stop due to the requirement that the decrypted message must begin with `HTTPReq`).

We also note that the encrypted message created by the authorization server containing the nonce also contains a public TLS key of as. (This holds true due to Lemma 1).

After decrypting the mTLS nonce and public TLS key of as, the client stores the sequence $\langle request.\texttt{host}, clientId, pubKey, mtlsNonce\rangle$ into the `mtlsCache` subterm of its state, where, in particular,

- $request$.`host` is a domain of as (see Line 5, Algorithm 2)
- $mtlsNonce$ is the mTLS nonce chosen by as.

Thus, the nonce is stored at the client together with a domain of the authorization server. After storing the values, the client stops in Line 9 of Algorithm 2 without creating an event and without storing the nonce in any other place.

c **sends mTLS nonces only to domains of** as**.** The client accesses values stored in the `mtlsCache` subterm of its state only in the following places:

Case 1: Algorithm 3

In this algorithm, the client accesses the `mtlsCache` subterm only in Line 14 and Line 27.

In both cases, the sequence containing the nonce is removed from the `mtlsCache` subterm (Lines 16 and 28), and the mTLS nonce is sent by calling the HTTPS_SIMPLE_SEND function. The HTTP request that is passed to HTTPS_SIMPLE_SEND in Line 41 contains the retrieved mTLS nonces only in the body, under the dictionary key `TLS_AuthN` (Line 15, Line 39) or `TLS_binding` (Line 25, Line 29, Line 39).

In all cases, the domain stored in the sequence that is retrieved from the `mtlsCache` subterm of the client state (i.e., the first entry of the sequence) is the host of the HTTPS request that the client constructs (see Lines 14, 27).

Case 2: Algorithm 4

Here, the client accesses the `mtlsCache` state subterm only in Line 13. As in the first case, the sequence from which the mtls nonce is chosen is removed from the `mtlsCache` subterm (Line 16 of Algorithm 4). The nonce is sent in the body of an HTTP request, using the dictionary key `TLS_binding` (see Line 14) by calling HTTPS_SIMPLE_SEND in Line 30. The request is sent to the same domain that is stored in the sequence containing the mtls nonce.

Case 3: Algorithm 6

Here, Line 19 is the last line in which the client accesses the `mtlsCache` state subterm. As in the previous cases, the client removes the corresponding sequence from the `mtlsCache` subterm (Line 21). The client creates an HTTPS request which contains the mTLS nonce in the body under the key `TLS_AuthN` (Lines 20, 33, and 35). Again, the request is sent to the same domain that is stored in the sequence containing the mtls nonce (see Line 35).

In all cases, the HTTP request is sent to the domain stored in the first sequence entry of the sequence containing the mTLS nonce (stored in the `mtlsCache` subterm). The request is sent by calling the HTTPS_SIMPLE_SEND function provided by the generic HTTPS server.

as **does not leak mTLS nonce contained in request.** The HTTPS request created by the client c using HTTPS_SIMPLE_SEND is encrypted asymetrically with a key of as. The authorization server only decrypts terms in the generic HTTPS server algorithms. More specifically, this request is decrypted (only) in Line 7 of Algorithm 31, as this is the only place where an authorization server decrypts a message asymetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 9.

In Algorithm 9, none of the endpoints except for the PAR (Line 53) and token endpoint (Line 81) reads, stores, or sends out a value stored in the body of the request under the `TLS_AuthN` or `TLS_binding` key.

The PAR and token endpoints pass the HTTP request to the AUTHENTICATE_CLIENT helper function (Algorithm 10), which removes an entry from the `mtlsRequests` state subterm and returns this entry; the `/par` endpoint code does not use this value. The token endpoint uses this value for token binding (Lines 110–120), but the nonce is not added to any state subterm and not sent out in a network message. Thus, the endpoints of the authorization server do not store the mTLS nonces contained in requests in any state subterm and do not send them out in any network message.

c **does not leak mTLS nonce in request after getting the response.** When receiving the HTTPS response, the generic HTTPS server removes the message from the $pendingRequests$ state subterm and calls PROCESS_HTTPS_RESPONSE with the request as the third function argument. Algorithm 2 does not store a nonce contained in the body of the request and does not create new network messages containing such a nonce.

Summing up, the client sends the mTLS nonce created by the authorization server only back to that authorization server. As an honest authorization server never sends out such a nonce received in a token request, we conclude that the nonce never leaks to any other process, in particular not to p. ■

Lemma 6 (mTLS Nonce created by RS does not Leak). For every run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every resource server $rs \in \mathsf{RS}$ that is honest in S^n, every client $c \in \mathsf{C}$ that is honest in S^n, every domain $d_c \in \mathsf{dom}(c)$, every $i \in \mathbb{N}$ with $0 \leq i \leq |S(rs).\texttt{mtlsRequests}|$ and with $S(rs).\texttt{mtlsRequests}.i.2 \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$, and every process p with $rs \neq p \neq c$ it holds true that $mtlsNonce := S(rs).\texttt{mtlsRequests}.i.1$ does not leak to p, i.e., $mtlsNonce \notin d_\emptyset(S^n(p))$.

PROOF. This proof is similar to the proof of Lemma 5:

Initially, the `mtlsRequests` subterm of the resource server's state is empty, i.e., $S^0(rs).\texttt{mtlsRequests} \equiv \langle\rangle$ (Definition 13). A resource server only adds values to the `mtlsRequests` subterm in Line 5 of Algorithm 12, where the mTLS nonce (the first value of the sequence that is added to `mtlsRequests`) is a fresh nonce (Line 3 of Algorithm 12).

Let $(S^i, E^i, N^i) \rightarrow (S^{i'}, E^{i'}, N^{i'})$ be the processing step in which the nonce is chosen (note that (S^i, E^i, N^i) is prior to (S, E, N) in ρ). In the same processing step, the resource server sends out the nonce in Line 7 of Algorithm 12, asymmetrically encrypted with the public key $\mathsf{pub}(\mathsf{tlskey}(d_c))$ (precondition of the lemma, see also Line 5 and Line 6 of Algorithm 12).

The corresponding private key is $\mathsf{tlskey}(d_c) \in tlskeys^c$, which is only known to c (Lemma 2). (The $mtlsNonce$ saved in `mtlsRequests` is not sent in any other place).

This implies that the encrypted nonce can only be decrypted by c. c decrypts messages only in Line 3 of Algorithm 2. (The only other place where a message is decrypted asymmetrically by c is in the generic HTTPS server (Line 7 of Algorithm 31), where the process would stop due to the requirement that the decrypted message must begin with `HTTPReq`).

We also note that the encrypted message created by the resource server containing the nonce also contains a public TLS key of rs. (This holds true due to Lemma 1).

After decrypting the mTLS nonce and public TLS key of rs, the client stores the sequence $\langle request.\texttt{host}, clientId, pubKey, mtlsNonce\rangle$ into the `mtlsCache` subterm of its state, where, in particular,

- $request.\texttt{host}$ is a domain of rs (see Line 5, Algorithm 2)
- $mtlsNonce$ is the mTLS nonce chosen by rs.

Thus, the nonce is stored at the client together with a domain of the resource server. After storing the values, the client stops in Line 9 of Algorithm 2 without creating an event and without storing the nonce in any other place.

c sends mTLS nonces only to domains of rs. The client accesses values stored in the `mtlsCache` subterm of its state only in the following places:

Case 1: Algorithm 3

In this algorithm, the client accesses the `mtlsCache` subterm only in Line 14 and Line 27.

In both cases, the sequence containing the nonce is removed from the `mtlsCache` subterm (Lines 16 and 28), and the mTLS nonce is sent by calling the HTTPS_SIMPLE_SEND function. The HTTP request that is passed to HTTPS_SIMPLE_SEND in Line 41 contains the retrieved mTLS nonces only in the body, under the dictionary key `TLS_AuthN` (Line 15, Line 39) or `TLS_binding` (Line 25, Line 29, Line 39).

In all cases, the domain stored in the sequence that is retrieved from the `mtlsCache` subterm of the client state (i.e., the first entry of the sequence) is the host of the HTTPS request that the client constructs (see Lines 14, 27).

Case 2: Algorithm 4

Here, the client accesses the `mtlsCache` state subterm only in Line 13. As in the first case, the sequence from which the mtls nonce is chosen is removed from the `mtlsCache` subterm (Line 16 of Algorithm 4). The nonce is sent in the body of an HTTP request, using the dictionary key `TLS_binding` (see Line 14) by calling HTTPS_SIMPLE_SEND in Line 30. The request is sent to the same domain that is stored in the sequence containing the mtls nonce.

Case 3: Algorithm 6

Here, Line 19 is the last line in which the client accesses the `mtlsCache` state subterm. As in the previous cases, the client removes the corresponding sequence from the `mtlsCache` subterm (Line 21). The client creates an HTTPS request which contains the mTLS nonce in the body under the key `TLS_AuthN` (Lines 20, 33, and 35). Again, the request is sent to the same domain that is stored in the sequence containing the mtls nonce (see Line 35).

In all cases, the HTTP request is sent to the domain stored in the first sequence entry of the sequence containing the mTLS nonce (stored in the `mtlsCache` subterm). The request is sent by calling the HTTPS_SIMPLE_SEND function provided by the generic HTTPS server.

rs does not leak mTLS nonce contained in request. The HTTPS request created by the client c (in one of the three aforementioned cases) using HTTPS_SIMPLE_SEND is encrypted asymetrically with a key of rs. The resource server only decrypts terms in the generic HTTPS server algorithms. More specifically, this request is decrypted (only) in Line 7 of Algorithm 31, as this is the only place where an resource server decrypts a message asymetrically, and then used as a function argument for PROCESS_HTTPS_REQUEST which is modeled in Algorithm 12.

In Algorithm 12, the `/MTLS-prepare` and `/DPoP-nonce` endpoints (Line 2 and Line 8 of Algorithm 12) do not read, store, or send out a value stored in the body of the request under the `TLS_AuthN` or `TLS_binding` key.

The last endpoint starting at Line 13 of Algorithm 12 accesses values stored in the body of the request under the `TLS_binding` key in Line 23. This value is not added to any state subterm and not sent out in a network message if Line 57 of Algorithm 12 is executed. If Line 48 of Algorithm 12 is true, then the whole request (including the `TLS_binding` value in the request body) is stored in the `pendingResponses` subterm of the resource server's state. However, the resource server never stores the body of requests stored in `pendingResponses` into any other subterm of its state and does not send out any value contained in the body.

c **does not leak mTLS nonce in request after getting the response.** When receiving the HTTPS responseto the request that the client creates in one of the three cases, the generic HTTPS server removes the message from the $pendingRequests$ state subterm and calls PROCESS_HTTPS_RESPONSE with the request as the third function argument. Algorithm 2 does not store a nonce contained in the body of the request and does not create new network messages containing such a nonce.

Summing up, the client sends the mTLS nonce created by the resource server only back to that resource server. As an honest resource server never sends out such a nonce received in a request, we conclude that the nonce never leaks to any other process, in particular not to p. ■

Lemma 7 (JWS client assertion created by client does not leak). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S with client identifier $clientId$ at as, every term t with

- $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(c))) \equiv \top$
- $\mathsf{extractmsg}(t)[\texttt{iss}] \equiv clientId$
- $\mathsf{extractmsg}(t)[\texttt{sub}] \equiv clientId$
- $\mathsf{extractmsg}(t)[\texttt{aud}].\texttt{host} \in \mathsf{dom}(as)$ or $\mathsf{extractmsg}(t)[\texttt{aud}] \in \mathsf{dom}(as)$

and every process p with $as \neq p \neq c$ it holds true that $t \notin d_\emptyset(S(p))$.

PROOF. The private signing key $\mathsf{signkey}(c)$ is part of the clients initial state ($s_0^c.\texttt{jwk}$, see Definition 11). Initially, no other process and no waiting event contains the key except as a public key $\mathsf{pub}(\mathsf{signkey}(c))$. The client c never leaks the private key: The client uses the private key only for creating signatures or creating public keys (from which the private key cannot be extracted) see Algorithm 3 (Line 19, 35, and 37), Algorithm 4 (Line 24 and 26), and Algorithm 6 (Line 24).

Thus, only c can create a term t, i.e., the attacker cannot create t itself by signing a dictionary with the corresponding `iss`, `sub`, and `aud` values. In the following, we show that such a term created by c does not leak to the attacker.

The client signs dictionaries with `aud` dictionary key only in two locations:

Case 1: Line 19 of Algorithm 3 The signature created in Line 19 of Algorithm 3 is added to the body of an HTTP request (Lines 20, 39, and 40). The client sends the HTTP request to $\mathsf{extractmsg}(t)[\texttt{aud}].\texttt{host}$ via HTTPS_SIMPLE_SEND in Line 41 (using `responseTo`: `TOKEN` in the first function argument).

Case 2: Line 24 of Algorithm 6 As in the previous case, the signature created in Line 24 of Algorithm 6 is added to the body of an HTTP request (Lines 25, 33, and 35). The client sends the HTTP request to $\mathsf{extractmsg}(t)[\texttt{aud}].\texttt{host}$ via HTTPS_SIMPLE_SEND in Line 36 (using `responseTo`: `PAR` in the first function argument).

We note that the client never signs a dictionary in which the `aud` value is a domain.

When the client receives the HTTPS response, the generic HTTPS server decrypts the message and calls PROCESS_HTTPS_RESPONSE. The original request (containing the client assertion) is used as the third function argument. The instantiation of PROCESS_HTTPS_RESPONSE (Algorithm 2) does not access the body of the request when processing `TOKEN` or `PAR` responses.

When processing the HTTPS request in Algorithm 9, the authorization server does not store the client assertion and does not create a network message containing the client assertion.

Overall, we conclude that no other process can derive a client assertion created by an honest client for an honest authorization server. ■

Lemma 8 (Client Authentication). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every authorization server $as \in \mathsf{AS}$, every client $c \in \mathsf{C}$, every processing step Q in ρ

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

with c and as being honest in S' and every client identifier $clientId$ of c at as it holds true that:

If $e_{\text{in}} \equiv \langle x, y, \mathsf{enc_a}(\langle m, k \rangle, k') \rangle$ (for some x, y, k, k') with m being an HTTP request such that all of the following hold true, then c *created* m (Definition 72):

- $\texttt{client\_id} \in m.\texttt{body} \Rightarrow m.\texttt{body}[\texttt{client\_id}] \equiv clientId$ and
- $\texttt{client\_assertion} \in m.\texttt{body} \Rightarrow \mathsf{extractmsg}(m.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}] \equiv clientId$ and
- $m.\texttt{path} \equiv /\texttt{par} \vee m.\texttt{path} \equiv /\texttt{token}$ and
- E_{out} is not empty.

PROOF. We first note that authorization servers do not emit any HTTP(S) requests.

as **executes either Line 80 or Line 142 of Algorithm 9.** We first show that in processing step Q, as executes either Line 80 or Line 142 of Algorithm 9: When processing e_{in}, the generic HTTPS server calls PROCESS_HTTPS_REQUEST in Line 9 of Algorithm 31, as the checks done in Line 2, 10, 19 and 25 complete successfully and the authorization server's

instantiation of PROCESS_OTHER (Algorithm 30) does not create an output event, i.e., if as ended up in Algorithm 30, E_{out} would be empty. Furthermore, the **stop** in Line 8 of Algorithm 31 is not executed as this **stop** does not emit an output event.

Thus, Algorithm 31 calls PROCESS_HTTPS_REQUEST in processing step Q. The authorization server's instantiation of PROCESS_HTTPS_REQUEST is defined in Algorithm 9.

If $m.\texttt{path} \equiv \texttt{/par}$ (with m as in the statement of the lemma), then the PAR endpoint starting in Line 53 of Algorithm 9 is executed. No **stop** except for the last (unconditional) stop in Line 80 emits an event.

Analogously, if $m.\texttt{path} \equiv \texttt{/token}$ (with m as in the statement of the lemma), then the (unconditional) **stop** in Line 142 was executed in the processing step, as no other **stop** emits events.

HTTP request contains values that only c and as know. Non-empty E_{out} implies that the checks done in the AUTHENTICATE_CLIENT helper function (Algorithm 10, called in Line 56 or 84 of Algorithm 9) did not lead to a **stop**.

In both cases, Algorithm 10 is called with the HTTP request m and $S(as)$ as input arguments. As Line 26 of Algorithm 10 is not executed (because E_{out} is not empty), it follows that $\texttt{client\_assertion} \in m.\texttt{body}$ or $\texttt{TLS\_AuthN} \in m.\texttt{body}$.

Case 1: $\texttt{client\_assertion} \in m.\texttt{body}$**:** As $\mathsf{extractmsg}(m.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}] \equiv \mathit{clientId}$ (lemma precondition), it holds true that the verification key used for verifying the signature in Line 4 of Algorithm 10 is

$$
\begin{aligned}
& S(as).\texttt{clients}[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Line 4 and 12, Algorithm 10)} \\
\equiv\ & s_0^{as}.\texttt{clients}[\mathit{clientId}][\texttt{jwk\_key}] && \text{(value is never changed)} \\
\equiv\ & \mathsf{clientInfoAS}(as)[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Definition 12)} \\
\equiv\ & \langle\{\langle \mathit{cli}.\texttt{client\_id}, \mathsf{as\_cli}(\mathit{cli})\rangle \mid d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as)\}\rangle[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Definition 9)}
\end{aligned}
$$

with $\mathit{cli} = \mathsf{clientInfo}(d_{as}, d_c)$ and as_cli as in Definition 9. As $\mathit{clientId}$ is the identifier of c at as and as no two clients have the same identifier at an authorization server (see Definition 7), it follows that $d_c \in \mathsf{dom}(c)$.

As there is a dictionary entry in $S(as).\texttt{clients}[\mathit{clientId}]$ with the key jwk_key (otherwise, as would **stop** with empty E_{out} in Line 5 of Algorithm 10), it follows that the verification key is

$$
\begin{aligned}
& \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{jwtSkey}) && (d_{as} \in \mathsf{dom}(as),\ d_c \in \mathsf{dom}(c);\ \text{see also Definition 9}) \\
\equiv\ & \mathsf{pub}(\mathsf{signkey}(c)) && \text{(Definition 7)}
\end{aligned}
$$

The corresponding private key $\mathsf{signkey}(c)$ is only known to c (Lemma 2).

In the following, let $\mathit{cli\_assertion} = \mathsf{extractmsg}(m.\texttt{body}[\texttt{client\_assertion}])$. As $\mathit{cli\_assertion}[\texttt{iss}] \equiv \mathit{clientId}$ (precondition of the lemma), $\mathit{cli\_assertion}[\texttt{sub}] \equiv \mathit{clientId}$ (Line 12 of Algorithm 10), and $\mathit{cli\_assertion}[\texttt{aud}].\texttt{host} \in \mathsf{dom}(as)$ or $\mathit{cli\_assertion}[\texttt{aud}] \in \mathsf{dom}(as)$ (Line 14 of Algorithm 10 and as the host of the request is a domain of the authorization server as shown in Lemma 1), we can apply Lemma 7.

Thus, for all processes p, it holds true that $m.\texttt{body}[\texttt{client\_assertion}] \notin d_\emptyset(S'(p))$ if $as \neq p \neq c$, i.e., only c and as can derive $m.\texttt{body}[\texttt{client\_assertion}]$. As authorization servers do not create HTTP requests, it follows that m was *created* by c.

Case 2: $\texttt{TLS\_AuthN} \in m.\texttt{body}$**:** From Lines 17–19 of Algorithm 10 it follows that

$$\exists i \in \mathbb{N}.\ S(as).\texttt{mtlsRequests}[m.\texttt{body}[\texttt{client\_id}]].i.1 \equiv m.\texttt{body}[\texttt{TLS\_AuthN}]$$

Note that $\texttt{client\_id} \in m.\texttt{body}$ as otherwise, the **stop** in Line 23 of Algorithm 10 will be executed.

Now, we can apply Lemma 5 with ρ' (ρ' being the trace prefix of ρ up to and including (S', E', N')).

Thus, for all processes p, it holds true that $m.\texttt{body}[\texttt{TLS\_AuthN}] \notin d_\emptyset(S'(p))$ if $as \neq p \neq c$, i.e., only c and as can derive $m.\texttt{body}[\texttt{TLS\_AuthN}]$. As authorization servers do not create HTTP requests, we conclude that m was *created* by c.

■

Lemma 9 (ID tokens do not leak). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S with client identifier $\mathit{clientId}$ at as, every term t with

- $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(as))) \equiv \top$
- $\mathsf{extractmsg}(t)[\texttt{aud}] \equiv \mathit{clientId}$

and every process p with $as \neq p \neq c$ it holds true that $t \notin d_\emptyset(S(p))$.

PROOF. An honest authorization server creates signatures with an aud value only in Line 139 of Algorithm 9. The only other location where an authorization server creates signatures is Line 127 of Algorithm 9, where the term does not contain an aud attribute. Together with Lemma 4 we can conclude that t was generated in Line 139 of Algorithm 9.

Let m_{req} be the corresponding request that was processed at the /token endpoint (starting at Line 81 of Algorithm 9). The token endpoint of the authorization server calls the function AUTHENTICATE_CLIENT (Line 84), which returns a sequence $authnResult$. Due to Lines 84, 93 and 136 of Algorithm 9, it holds true that $authnResult.1 \equiv clientId$.

AUTHENTICATE_CLIENT (Algorithm 10) is called with m_{req} as the first function argument and ensures that the first value of the returned sequence is either

- $\mathsf{extractmsg}(m_{\text{req}}.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}]$ (Lines 3, 11, and 12 of Algorithm 10), or
- $m_{\text{req}}.\texttt{body}[\texttt{client\_id}]$ (Line 17 of Algorithm 10).

m_{req} is the input argument of the PROCESS_HTTPS_REQUEST function (Algorithm 9), which is called only in Line 9 of the generic HTTPS server model (Algorithm 31), it follows that the input event of Algorithm 31 has the form $\langle x, y, \mathsf{enc_a}(\langle m_{\text{input}}, k\rangle, k')\rangle$ (for some values x, y, k, k'), with

- $m_{\text{input}}.\texttt{body}[\texttt{client\_id}] \equiv clientId$ or
- $\mathsf{extractmsg}(m_{\text{input}}.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}] \equiv client\_id$
- $m_{\text{input}}.\texttt{path} \equiv \texttt{/token}$

As shown in Lemma 8, the event $\langle x, y, \mathsf{enc_a}(\langle m_{\text{input}}, k\rangle, k')\rangle$ was created by c. The term created in Line 139 of Algorithm 9 is sent back to the sender of the request in Line 142 of Algorithm 9, encrypted with the symmetric key k. Thus, only c can decrypt the response.

As an honest client never sends out an id token, it follows that t does not leak to p.

■

Lemma 10 (mTLS Keys of Clients stored at Authorization Servers). For every authorization server $as \in \mathsf{AS}$ and every term $clientId \in \mathcal{T}_{\mathcal{N}}$ it holds true that if $s_0^{as}.\texttt{clients}[clientId][\texttt{mtls\_key}]$ is not $\mathsf{pub}(\diamond)$ and not $\langle\rangle$, then $\exists c \in \mathsf{C}, d \in \mathsf{dom}(c)$ such that $s_0^{as}.\texttt{clients}[clientId][\texttt{mtls\_key}] \equiv \mathsf{pub}(\mathsf{tlskey}(d))$.

PROOF.

$s_0^{as}.\texttt{clients}[clientId][\texttt{mtls\_key}]$

$\equiv \mathsf{clientInfoAS}(\mathsf{as})[clientId][\texttt{mtls\_key}]$ (Definition 12)

$\equiv \langle\{\langle cli.\texttt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid \exists\, d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(\mathsf{as}) \colon \mathsf{clientInfo}(d_{as}, d_c) \equiv cli\}\rangle[clientId][\texttt{mtls\_key}]$ (Definition 9)

$\equiv \mathsf{as\_cli}(\mathsf{clientInfo}(d_{as}, d_c))[\texttt{mtls\_key}]$ (with $d_c \in \mathsf{Doms}$, $d_{as} \in \mathsf{dom}(\mathsf{as})$, $\mathsf{clientInfo}(d_{as}, d_c) \equiv clientId$)

$\equiv \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{mtlsSKey})$ (as the value is not $\langle\rangle$ (precondition);
with $d_c \in \mathsf{Doms}$, $d_{as} \in \mathsf{dom}(\mathsf{as})$, $\mathsf{clientInfo}(d_{as}, d_c) \equiv clientId$;
see Definition 9 for the definition of as_cli)

As this value is not $\mathsf{pub}(\diamond)$ (precondition of the lemma), it follows from Definition 7 that the value is equal to $\mathsf{pub}(\mathsf{tlskey}(d_c))$ and d_c is a domain of a client $c \in \mathsf{C}$ (as clientInfos is defined for domains of clients, see Definition 7).

Lemma 11 (DPoP proof secrecy (RS)). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every resource server $rs \in \mathsf{RS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S, every term t with

- $\mathsf{checksig}(t, \mathsf{pub}(\mathsf{signkey}(c))) \equiv \top$
- $\mathsf{extractmsg}(t)[\texttt{payload}][\texttt{htu}].\texttt{host} \in \mathsf{dom}(rs)$,
- $\texttt{ath} \in^{\langle\rangle} \mathsf{extractmsg}(t)[\texttt{payload}]$,
- $\mathsf{extractmsg}(t)[\texttt{payload}][\texttt{nonce}] \in S(rs).\texttt{dpopNonces}$

and every process p with $rs \neq p \neq c$ it holds true that $t \notin d_\emptyset(S(p))$.

PROOF. The private signing key $\mathsf{signkey}(c)$ is part of the clients initial state ($s_0^c.\texttt{jwk}$, see Definition 11). Initially, no other process and no waiting event contains the key except as a public key $\mathsf{pub}(\mathsf{signkey}(c))$. The client c never leaks the private key: The client uses the private key only for creating signatures or creating public keys (from which the private key cannot be extracted) see Algorithm 3 (Line 19, 35, and 37), Algorithm 4 (Line 24 and 26), and Algorithm 6 (Line 24).

Thus, only c can create a term t, i.e., the attacker cannot create t itself by signing a dictionary with the corresponding payload value. In the following, we show that such a term created by c does not leak to the attacker.

The client signs dictionaries with a payload dictionary key only in two locations: In Line 37 of Algorithm 3, where the payload dictionary does not contain an ath value (see Line 36 of Algorithm 3), and in Line 26 of Algorithm 4.

In Line 26 of Algorithm 4, the client sends the term t to $\mathsf{extractmsg}(t)[\texttt{payload}][\texttt{htu}].\texttt{host}$ via HTTPS_SIMPLE_SEND (using responseTo: RESOURCE_USAGE in the first function argument), see Lines 21, 25, 29, and Line 30 of Algorithm 4. The client does not store t in any other subterm except for those needed by HTTPS_SIMPLE_SEND. The term t is added

(only) to the headers of the HTTP request using the DPoP dictionary key, see Line 28 of Algorithm 4. The client also adds an Authorization header containing a dictionary with a DPoP dictionary key, and in particular, no other value, see Line 27 and Line 29 of Algorithm 4.

When the client receives the HTTPS response to this request, the generic HTTPS server decrypts the message and calls PROCESS_HTTPS_RESPONSE. The original request (containing the signed term) is used as the third function argument. The instantiation of PROCESS_HTTPS_RESPONSE (Algorithm 2) does not access the headers of the request when processing RESOURCE_USAGE responses.

When processing the HTTPS request created by the client in Algorithm 12, the resource server does not access the DPoP header (in particular, it does not add the term to its state and does not create a network message containing the value) in the /MTLS-prepare and /DPoP-nonce endpoints (Lines 2 and Line 8 of Algorithm 12). For all other path values (Line 13 of Algorithm 12), the resource server first checks whether the resource identified by the path is managed by a supported authorization server. If this is not the case, then the resource server stops without changing the state and without emitting events (Line 17 of Algorithm 12). Otherwise, the resource server will eventually invalidate the nonce value stored in the DPoP proof in Line 42 of Algorithm 12 (by removing it from the dpopNonces subterm of the resource server's state), as the request contains an Authorization header containing a dictionary with the DPoP keyword (see Line 19 and Line 28 of Algorithm 12). The **stop**s before the removal of the nonce from the state of the resource server do not modify the state of the resource server and do not lead to new events.

We note that the dpopNonces state subterm of the resource server does not contain any value twice, as the resource server only adds fresh nonces to the state subterm, see the endpoint in Line 8 of Algorithm 12. Thus, the nonce is not contained in dpopNonces after Line 42 of Algorithm 12 is executed, and the resource server it does not add it back to the dpopNonces state subterm afterwards.

Thus, if the resource server does not finish with a **stop** without any arguments, it holds true that $\mathsf{extractmsg}(t)[\texttt{payload}][\texttt{nonce}]$ is not contained in the dpopNonces subterm of the new resource server's state, as it always stops with the updated state. (If it finishes with a **stop** without any arguments, then t will not leak, as there is no change in any state and no new event).

Overall, we conclude that no other process can derive a signed term t (as in the statement of the lemma) created by an honest client for an honest resource server.

■

Lemma 12 (Access Token can only be used by Honest Client). For

- every run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of $\mathcal{FAPI}$ with a network attacker,
- every resource server $rs \in \mathsf{RS}$ that is honest in S^n,
- every identity $id \in^{\langle\rangle} s_0^{rs}.\texttt{ids}$,
- every processing step in ρ

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \to E^Q_{\text{out}}]{e^Q_{\text{in}} \to rs} (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$$

- every $resourceID \in \mathbb{S}$ with $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ being honest in S^Q,

it holds true that:

If $\exists x, y, k, m_{\text{resp}}.\langle x, y, \mathsf{enc_s}(m_{\text{resp}}, k)\rangle \in^{\langle\rangle} E^Q_{\text{out}}$ such that m_{resp} is an HTTP response with $m_{\text{resp}}.\texttt{body}[\texttt{resource}] \in^{\langle\rangle} S^{Q\prime}(rs).\texttt{resourceNonce}[id][resourceID]$, then

1) There exists a processing step

$$P = (S^P, E^P, N^P) \xrightarrow[rs \to E^P_{\text{out}}]{e^P_{\text{in}} \to rs} (S^{P\prime}, E^{P\prime}, N^{P\prime})$$

such that

a) either $P = Q$ or P prior to Q in ρ, and
b) e^P_{in} is an event $\langle x, y, \mathsf{enc_a}(\langle m_{\text{req}}, k_1\rangle, k_2)\rangle$ for some x, y, k_1, and k_2 where $m_{\text{req}} \in \mathcal{T}_{\mathcal{N}}$ is an HTTP request which contains a term (access token) t in its Authorization header, i.e., $t \equiv m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$, and
c) r was generated in P in Line 46 of Algorithm 12.

2) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, as, and id in S^Q (see Definition 14).
3) If there exists a client $c \in \mathsf{C}$ such that $k \equiv \mathsf{pub}(\mathsf{signkey}(c))$ or $k \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$ and $d_c \in \mathsf{dom}(c)$, and if c is honest in S^n, then e^P_{in} was created by c.

PROOF. An honest resource server sends HTTPS responses with a resource dictionary key only in Line 69 of Algorithm 12 and Line 22 of Algorithm 13.

Case 1: Line 69 of Algorithm 12

First Postcondition In the same processing step, i.e., $P = Q$, the resource server received an HTTPS request with an access token and generated the resource:
e^Q_{in} is an event containing an HTTPS request, as Algorithm 12 is only called by the generic HTTPS server in Line 9 of Algorithm 31. As the check done in Line 7 of Algorithm 31 was true and the `stop` in Line 8 was not executed, it follows that the input event of Algorithm 31 was an event containing an HTTPS request m_{req} (as in the first statement of the post-condition of the lemma).
m_{req} contains an `Authorization` header (Line 19 of Algorithm 12).
The resource that is sent out in Line 69 of Algorithm 12 is a freshly chosen nonce generated in the same processing step in Line 46 of Algorithm 12 (see also Line 66 of Algorithm 12). This concludes the proof of the first post-condition.

Second Postcondition As Line 69 of Algorithm 12 is in the second case of Line 48 of Algorithm 12, it follows that $\mathsf{extractmsg}(m.\texttt{headers}[\texttt{Authorization}].2)$ is a structured access token (see Lines 21 and 47).
The access token is signed by $\mathsf{authorizationServerOfResource}^{rs}(resourceID)$: The value of $responsibleAS$ (in Line 15) is equal to

$$\begin{aligned}
& S^Q(rs).\texttt{resourceASMapping}[resourceID] && \text{(Line 15 of Algorithm 12)}\\
\equiv\ & s_0^{rs}.\texttt{resourceASMapping}[resourceID] && \text{(value is never changed)}\\
\in\ & \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID)) && \text{(Definition 13)}
\end{aligned}$$

As required by the precondition of the lemma, $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ is honest in S^Q.
The signature of the access token is checked in Line 60 of Algorithm 12 using the verification key

$$\begin{aligned}
& asInfo[\texttt{as_key}] && \\
\equiv\ & S^Q(rs).\texttt{asInfo}[responsibleAS][\texttt{as_key}] && (responsibleAS \in \mathsf{dom}(as), \text{Line 18})\\
\equiv\ & s_0^{rs}.\texttt{asInfo}[responsibleAS][\texttt{as_key}] && \text{(value is never changed)}\\
\equiv\ & \mathsf{signkey}(\mathsf{dom}^{-1}(responsibleAS)) && \text{(Definition 13)}\\
\equiv\ & \mathsf{signkey}(as) &&
\end{aligned}$$

The authorization server as only uses this key in the following locations:

- Line 13 of Algorithm 9: Endpoint returning public key
- Line 127 of Algorithm 9: Signing access token
- Line 139 of Algorithm 9: Signing ID token

As ID tokens created by an authorization server do not contain a `cnf` claim (see Lines 134-139 of Algorithm 9), it follows that $\mathsf{extractmsg}(m.\texttt{headers}[\texttt{Authorization}].2)$ is an access token created by as in Line 127 of Algorithm 9.
Let $O = (S^O, E^O, N^O) \xrightarrow[as \to E^O_{\text{out}}]{e^O_{\text{in}} \to as} (S^{O\prime}, E^{O\prime}, N^{O\prime})$ be the processing step in which the authorization server created and signed the access token. After finishing the processing step, as stores the access token in $S^{O\prime}(as).\texttt{records}.i[\texttt{access\_token}]$, for some natural number i (as Line 130 of Algorithm 9 was executed by the authorization server)
The structured access token contains a value $\mathsf{extractmsg}(m.\texttt{headers}[\texttt{Authorization}].2)[\texttt{sub}] \in^{\langle\rangle} S^Q(rs).\texttt{ids}$ (Line 47, 62, and 63 of Algorithm 12). This identity is used as a dictionary key for storing the resource (see Line 65 of Algorithm 12). The ids stored at the resource server are never changed, i.e., $S^Q(rs).\texttt{ids} \equiv s_0^{rs}.\texttt{ids}$. When creating the access token, the authorization server takes this value from $S^O(as).\texttt{records}.i[\texttt{sub}]$ with the same i as above (Line 92 and 126 of Algorithm 9). As the remaining lines of the token endpoint do not change this value, it follows that $S^O(as).\texttt{records}.i[\texttt{sub}] \equiv S^{O\prime}(as).\texttt{records}.i[\texttt{sub}]$.
From the successful check of Line 58 of Algorithm 12 (as we assume that the resource server returns a resource in Line 69), it follows that either

- $accessTokenContent[\texttt{cnf}].1 \equiv \texttt{x5t\#S256}$ or
- $accessTokenContent[\texttt{cnf}].1 \equiv \texttt{jkt}$,

as $cnfValue$ is set in Line 27 or Line 43 of Algorithm 12.
The authorization server sets the `cnf` value of access tokens only in Line 126 of Algorithm 9. The value is determined either in Line 109 or Line 120 of Algorithm 9, and the authorization server stores the `cnf` value into the same record as the `access_token` and `sub` values, see Line 131 of Algorithm 9, i.e., $S^{O\prime}(as).\texttt{records}.i[\texttt{cnf}]$ is either $[\texttt{jkt}: \mathsf{hash}(k)]$ or $[\texttt{x5t\#S256}: \mathsf{hash}(k)]$, for some value k.

As authorization servers do not remove sequences from their `records` state subterm, it follows that the access token is bound do some term $k \in \mathcal{T}_{\mathcal{N}}$, the authorization server as, and id in S^Q, by which we conclude the proof of the second postcondition for this case.

Third Postcondition

Let $c \in \mathsf{C}$ be honest in S^n.

Case 1.3.1: $k \equiv \mathsf{pub}(\mathsf{signkey}(c))$ As already shown, the authorization server determines the `cnf` value either in Line 109 or Line 120 of Algorithm 9.

- Line 109 of Algorithm 9: The structured access token contains the value $accessTokenContent[\texttt{cnf}].1 \equiv \texttt{jkt}$. Thus, the resource server executed Line 43 of Algorithm 12 (in the processing step P), i.e., the request e^P_{in} contains a DPoP proof $dpopProof$ (in the header of the request, see Line 29 of Algorithm 12). All preconditions of Lemma 11 are true:
  - $\mathsf{checksig}(dpopProof, \mathsf{pub}(\mathsf{signkey}(c))) \equiv \top$ (see Line 32 of Algorithm 12)
  - $\mathsf{extractmsg}(dpopProof)[\texttt{payload}][\texttt{htu}].\texttt{host} \in \mathsf{dom}(rs)$ (see Line 36 of Algorithm 12 and Lemma 1)
  - $\texttt{ath} \in^{\langle\rangle} \mathsf{extractmsg}(dpopProof)[\texttt{payload}]$, (see Line 40 of Algorithm 12)
  - $\mathsf{extractmsg}(dpopProof)[\texttt{payload}][\texttt{nonce}] \in S(rs).\texttt{dpopNonces}$ (see Line 38 of Algorithm 12)

  Thus, we can apply Lemma 11 and conclude that `dpopProof` can only be known by c and rs. As resource servers do not send requests to themselves, it follows that only c could have created the request e^P_{in}.
- Line 120 of Algorithm 9: The structured access token contains the value $accessTokenContent[\texttt{cnf}].1 \equiv \texttt{x5t\#S256}$.

  The value $accessTokenContent[\texttt{cnf}].2$ was chosen by the authorization server in Line 120 of Algorithm 9 (as this is the only location where the authorization server sets the `cnf` value with the `x5t#S256` dictionary key), where it was set to $\mathsf{hash}(mTlsKey)$. The value $mTlsKey$ is set to $mtlsInfo.2$ in Line 119 of Algorithm 9. The sequence $mtlsInfo$ is chosen in Line 84 or Line 117 of Algorithm 9. In both cases, $mTlsKey$ is set to $s_0^{\mathsf{as}}.\texttt{clients}[clientId][\texttt{mtls\_key}]$:
  - Line 84 of Algorithm 9: $mtlsInfo$ is the third entry of the return value of AUTHENTICATE_CLIENT. As shown above, AUTHENTICATE_CLIENT (Algorithm 10) determines the client identifier $clientId$ from the HTTP request and also determines the type of the client (see Lines 7, 8, 20, 21). As shown previously, the type of the client is either `pkjwt_mTLS` or `mTLS_mTLS`. Thus, the body of the request does not contain a value `client_assertion`, as otherwise, the **stop** in Line 10 of Algorithm 10 would have prevented the authorization server to issue the access token. In particular, the **return** in Line 28 was executed and the third return value was taken from $S^O(as).\texttt{mtlsRequests}[clientId]$ (Line 19 of Algorithm 10). Initially, the `mtlsRequests` subterm of the authorization server's state is empty (see Definition 12). The authorization server adds values to `mtlsRequests` only in Line 161 of Algorithm 9. The second sequence entry is $s_0^{\mathsf{as}}.\texttt{clients}[clientId][\texttt{mtls\_key}]$ (due to Line 158 of Algorithm 9 and because the `clients` subterm of the authorization server's state is never modified).
  - Line 117 of Algorithm 9: $mtlsInfo$ is taken from $S^O(as).\texttt{mtlsRequests}[clientId]$. As shown in the previous case, the second sequence entry of $mtlsInfo$ is equal to $s_0^{\mathsf{as}}.\texttt{clients}[clientId][\texttt{mtls\_key}]$.

  When adding values to `mtlsRequests` in Line 161 of Algorithm 9, the authorization server ensures that the value of $s_0^{\mathsf{as}}.\texttt{clients}[clientId][\texttt{mtls\_key}]$ is not $\langle\rangle$ and not $\mathsf{pub}(\diamond)$ (Line 159 of Algorithm 9). Thus, we can apply Lemma 10 and conclude that there exists a client $c \in \mathsf{C}$ and $s_0^{\mathsf{as}}.\texttt{clients}[clientId][\texttt{mtls\_key}] \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$ with $d_c \in \mathsf{dom}(c)$.

  Thus, it is not possible that this value is a public signature verification key.

Case 1.3.2: $k \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$ **and** $d_c \in \mathsf{dom}(c)$ As in the previous case, the authorization server determined the `cnf` value either in Line 109 or Line 120 of Algorithm 9.

- Line 109 of Algorithm 9: As only c knows the private key $\mathsf{tlskey}(d_c)$, it follows that c created a term $dpopProof$ such that $\mathsf{checksig}(dpopProof, \mathsf{pub}(\mathsf{tlskey}(d_c))) \equiv \top$ (Line 103 of Algorithm 9). However, as c is an honest client, it never creates such a term, as all signatures are created using $\mathsf{signkey}(\texttt{c})$ (see Line 19 of Algorithm 3, Line 37 of Algorithm 3, Line 26 of Algorithm 4, Line 37 of Algorithm 3, Line 26 of Algorithm 4, Line 24 of Algorithm 6, and Definition 11).
- Line 120 of Algorithm 9: The structured access token contains the value $accessTokenContent[\texttt{cnf}].1 \equiv \texttt{x5t\#S256}$. Thus, the resource server executed Line 27 of Algorithm 12 (in the processing step P). This means that e^P_{in} contains a value $mtlsNonce$ in the body of the request such that $\langle mtlsNonce, \mathsf{pub}(\mathsf{tlskey}(d_c))\rangle \in^{\langle\rangle} S^P(rs).\texttt{mtlsRequests}$.

If the client c is honest in S^n, then it is also honest in S^P, and we can apply Lemma 6 and conclude that only c and rs can derive $m_{\text{req}}.\texttt{body}[\texttt{TLS\_binding}]$. As resource servers do not send requests containing `TLS_binding` in the request body, it follows that the HTTP request m_{req} was created by c.

Case 2: Line 22 of Algorithm 13

First Postcondition In Line 22 of Algorithm 13, the resource server is processing an HTTP response $resp_{\text{introsp}}$ (with the reference `TOKENINTROSPECTION`, see Line 2 of Algorithm 13). An honest resource server sends HTTP requests only by calling HTTPS_SIMPLE_SEND in Line 56 of Algorithm 12 (again with the reference `TOKENINTROSPECTION`). Let req_{introsp} be the corresponding request to $resp_{\text{introsp}}$. The processing step in which the resource server emitted req_{introsp} is P (as in the postcondition of the lemma): The input event of P contains an HTTP request m_{req} (again as in the first postcondition) with an access token $t \equiv m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$ (Line 19 of Algorithm 12). The resource r that the resource server sends out in Line 22 of Algorithm 13 (in the processing step Q) was stored by the resource server in $S^{P'}\texttt{pendingResponses}$ in Line 50 of Algorithm 12, and the resource was generated in Line 46 of Algorithm 12 (in the processing step P).

Second Postcondition The request req_{introsp} was sent by rs to a domain of as: $responsibleAS$ in Line 15 of Algorithm 12 is a domain of as, as shown in the proof of the first case. Thus, it follows that $S^P(rs).\texttt{asInfo}[responsibleAS][\texttt{as\_introspect\_ep}]$ is $\langle \texttt{URL}, \texttt{S}, dom_{as}, /\texttt{introspect}, \langle\rangle, \bot\rangle$, with $dom_{as} \in \mathsf{dom}(as)$ (see Definition 13).

Furthermore, req_{introsp} contains the value $m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$, see Line 21 and Line 54 of Algorithm 12. The authorization server as processes this request in the introspection endpoint in Line 143 of Algorithm 9. As the resource server did not stop in Line 11 of Algorithm 13, we conclude that the access token sent by the resource server in P is active, i.e., the authorization server executed Line 152 of Algorithm 9. Thus, there is a value $record$ in the `records` state subterm of the authorization server's state with the access token (Line 148 of Algorithm 9), and in this record, there is a `cnf` and a `subject` entry (Line 152 of Algorithm 9) The `cnf` and `subject` values are added to the body of the introspection response, and the resource server checks that the subject value is contained in the list of ids that the resource server stores in $S^Q(rs).ids$ (Line 16 of Algorithm 13).

An honest authorization server adds `cnf` values to an entry of its `records` state entry only in the token endpoint in Line 131 of Algorithm 9. Thus, this value is either $[\texttt{jkt}\colon \mathsf{hash}(k)]$ (see Line 109 of Algorithm 9), or $[\texttt{x5t\#S256}\colon \mathsf{hash}(k)]$ (see Line 120 of Algorithm 9), for some value k.

Third Postcondition The resource server checks in Line 13 of Algorithm 13 that the `cnf` value that the authorization server put into the response $resp_{\text{introsp}}$ is equal to the $cnfValue$ that the resource server stored in Line 50 of Algorithm 12 in the processing step P. The resource server does the same checks in P as in the first case (i.e., when sending out the response in Line 69 of Algorithm 12). Thus, it holds true that the request processed in P either contains a DPoP proof that only c and rs can derive, or an mTLS nonce that only c and rs can derive. The proof is analogous to the proof of the first case, i.e., only c could have created the request e^P_{in}.

■

Lemma 13 (Redirect URI Properties). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S with client identifier $clientId \not\equiv \langle\rangle$ at as, and every $requestUri$, all redirect URIs for c stored at as are HTTPS URIs and belong to c. Or, more formally: Let $rec = S(as).\texttt{authorizationRequests}[requestUri]$, then $rec[\texttt{client\_id}] \equiv cliendId$ implies both $rec[\texttt{redirect\_uri}].\texttt{protocol} \equiv \texttt{S}$, and $rec[\texttt{redirect\_uri}].\texttt{host} \in \mathsf{dom}(c)$

PROOF. The only places in which an honest authorization server writes to its `authorizationRequests` state subterm are:

- Line 28 of Algorithm 9: Here, the authorization server does not change or create values under the `client_id` or `redirect_uri` keys.
- Line 78 of Algorithm 9: See below.

In the latter case, the authorization server is processing a pushed authorization request, i.e., an HTTPS request req to the /par endpoint.

In order to get to Line 78 of Algorithm 9, req must contain valid client authentication data (see Lines 56 and 60), in particular, $req.\texttt{body}$ must contain a client id (under key `client_id`) and either a value under key `TLS_AuthN` or `client_assertion`. In the latter case, Line 4 of Algorithm 10 together with Line 12 of Algorithm 10 and Line 60 of Algorithm 9 ensure that $\mathsf{extractmsg}(req.\texttt{body}[\texttt{client\_assertion}])[\texttt{iss}] \equiv req.\texttt{body}[\texttt{client\_id}]$. We note that reaching Line 78 of Algorithm 9 implies that the current processing step will output an event (there are no **stop**s between Line 78 and Line 80 of Algorithm 9). Hence, we can apply Lemma 8.

When reaching Line 78 of Algorithm 9, req also must contain a $redirectUri$ value in $req.\texttt{body}[\texttt{redirect\_uri}]$ (Line 63 of Algorithm 9). Furthermore, this $redirectUri$ must be an HTTPS URI (Line 65 of Algorithm 9) and this is the value stored in

the authorization server's `authorizationRequests` state subterm (in a record under the key `redirect_uri`), together with $req.\texttt{body}[\texttt{client\_id}]$ (under key `client_id`).

Line 54 of Algorithm 9 ensures that $req.\texttt{body}$ contains a field `code_challenge_method` with value `S256`.

From Lemma 8, we know that c must have created req. Since c is honest and the only place in which an honest client produces an HTTPS request with a `code_challenge_method` with value `S256` is in Line 35 of Algorithm 6, we can conclude that the value of $req.\texttt{body}[\texttt{redirect\_uri}]$ is the one selected in Lines 2f. of Algorithm 6. This implies $req.\texttt{body}[\texttt{redirect\_uri}].\texttt{host} \in \mathsf{dom}(c)$ and hence $rec[\texttt{redirect\_uri}].\texttt{host} \in \mathsf{dom}(c)$. ■

Lemma 14 (Authorization code secrecy). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every configuration (S, E, N) in ρ, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ that is honest in S with client identifier $clientId$ at as, every identity $id \in \mathsf{ID}^{as}$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S, every authorization code $code \not\equiv \bot$ for which there is a record $rec \in^{\langle\rangle} S(as).\texttt{records}$ with $rec[\texttt{code}] \equiv code$, $rec[\texttt{client\_id}] \equiv clientId$, and $rec[\texttt{subject}] \equiv id$ and every process $p \notin \{as, c, b\}$, it holds true that $code \notin d_\emptyset(S(p))$.

PROOF.

1) For $code$ to end up in $\big(S(as).\texttt{records}.x\big)[\texttt{code}]$ (with $x \in \mathbb{N}$), the as has to execute Line 44 of Algorithm 9, since the only other places where an honest authorization server writes to the – initially empty, see Definition 12 – `records` state subterm are:
   - Line 123 of Algorithm 9: This line overwrites the stored authorization code with $\bot$, i.e., codes written by this line are not relevant to this lemma.
   - Line 130 of Algorithm 9 and Line 131 of Algorithm 9: In these two places, the authorization server does not modify the `code` entry. Note that ptr in these places cannot point "into" one of the records (see condition in Line 92 of Algorithm 9).
2) A $code$ stored in Line 44 of Algorithm 9 is a fresh nonce (Line 43 of Algorithm 9). Hence, a $code$ generated by as in that line in some processing step $s_i \to s_{i+1}$ is not known to any process up to and including s_i. Let e_{in} be the event processed by as in $s_i \to s_{i+1}$. In order to reach Line 44 of Algorithm 9, e_{in} must contain an HTTPS request req to the `/auth2` endpoint. The only place in which an honest as sends out the $code$ value is the HTTPS response to req – i.e., if the sender of req is honest, this response is only readable by the sender of req.
3) In addition, req must contain a valid $identity$–$password$ combination – because as stores $code$ along with $identity$ and $clientId$ only if $password \equiv \mathsf{secretOfID}(identity)$. Since as does not send requests to itself and $\mathsf{secretOfID}(identity)$ is only known to as and $\mathsf{ownerOfID}(identity)$, req must have been created by $\mathsf{ownerOfID}(identity)$ if the sender of req is honest. W.l.o.g., let $identity \equiv id$, i.e., req was created by b.
4) Since the origin header of req must be a domain of as and req must use the `POST` method, we know that req was initiated by a script of as. In particular, req must have been initiated by $script\_as\_form$ (as this is the only script ever sent by as). This script does not leak $code$ after it is returned from as, since it uses a form post to transmit the credentials to as, and the window is subsequently navigated away. Instead, as provides an empty script in its response to req (Line 52 of Algorithm 9). This response contains a location redirect header. It is now crucial to check that this redirect does not leak $code$ to any process except for c. The value of the location header is taken from $S(as).\texttt{authorizationRequests}[requestUri][\texttt{redirect\_uri}]$ where $S(as).\texttt{authorizationRequests}[requestUri][\texttt{client\_id}] \equiv clientId$. With Lemma 13, we have that this URI is an HTTPS URI and belongs to c. We therefore know that b will send an HTTPS request containing $code$ to c. We now have to check whether c or a script delivered by c to b will leak $code$. Algorithm 1 processes all HTTPS requests delivered to c. As as redirected b using the 303 status code, the request must be a GET request. Hence, c does not process this request in Lines 5ff. of Algorithm 1. If the request is processed in Lines 2ff. of Algorithm 1, c only responds with a script and does not use $code$ at all. This leaves us with Lines 12ff. of Algorithm 1; here, the $code$ value is (a) stored in the `sessions` state subterm and (b) given the SEND_TOKEN_REQUEST function. The value from (a) is not accessed anywhere, hence, it cannot leak. As for (b), we have to look at Algorithm 3. There, the $code$ is included in the body of an HTTPS request under the key `code` (Line 6 of Algorithm 3).
5) The HTTPS request ("token request") prepared in Lines 6ff. of Algorithm 3 is sent to the token endpoint of as (which was selected in b's initial request and is bound to the authorization response via the $\langle \texttt{\_\_Host}, \texttt{sessionId} \rangle$ cookie – see Line 13 of Algorithm 1 and Line 30 of Algorithm 2). Since an honest client does not change the contents of an element of `oauthConfigCache` once it is initialized with the selected authorization server's metadata (see Line 9 of Algorithm 6 and Line 16 of Algorithm 2), the token endpoint to which the $code$ is sent is the one provided by as at its metadata endpoint. As as is honest, the token endpoint returned by its metadata endpoint uses a domain which belongs to as and protocol `S`. Hence, the token request as such does not leak $code$.
6) As the token request is a HTTPS request sent to a domain of as and as is honest, only as can decrypt the request and extract $code$. Requests to the token endpoint are processed in Lines 81ff. of Algorithm 9, It is easy to see that the $code$ is not stored or send out there, hence, it cannot leak. ■

Lemma 15 (Request URIs do not Leak). For any run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, every client identifier $clientId$, every authorization server $as \in \mathsf{AS}$ that is honest in S, every client $c \in \mathsf{C}$ with client identifier $clientId$ at as that is honest in S, every browser $b \in \mathsf{B}$ that is honest in S, every domain $d_c \in \mathsf{dom}(c)$, every login session id $lsid$, every nonce $codeVerifier$ with

(a) $\langle\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle lsid, \top, \top, \top\rangle\rangle \in S(b).\texttt{cookies}[d_c]$, and
(b) $S(c).\texttt{sessions}[lsid][\texttt{code\_verifier}] \equiv codeVerifier$, and
(c) $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}] \in \mathsf{dom}(as)$, and
(d) c does not leak the authorization request for $lsid$ (see Definition 22),

then all of the following hold true

1) there is exactly one nonce $requestUri$, such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$, and
2) only b, c, and as know $requestUri$, i.e., for all processes $p \notin \{b, c, as\}$, we have $requestUri \notin d_\emptyset(S(p))$.

PROOF. We start by noting that all requests and responses at an authorization server's pushed authorization request (PAR) endpoint must be HTTPS requests, i.e., as long as the sender of the request and the authorization server in question are honest, the contents of request and response are not leaked by these messages as such (they may still leak by other means).

(I) $\mathsf{hash}(codeVerifier)$ **does not leak.** We start off by showing that $\mathsf{hash}(codeVerifier)$ does not leak to any process other than c and as. For this, we look at how $codeVerifier$ (from (b)) is generated and stored by c. The only place in which an honest client – such as c – stores a value under key `code_verifier` in its session storage is in PREPARE_AND_SEND_PAR in Line 34 of Algorithm 6. That value is generated in the same function in Line 26 by using a fresh nonce. Hence, at this point, $\mathsf{hash}(codeVerifier)$ is only derivable by c. PREPARE_AND_SEND_PAR ends with the client sending a PAR request which contains $\mathsf{hash}(codeVerifier)$ under the key `code_challenge`. So we have to check who receives that request. The PAR request is sent to the pushed authorization request endpoint of the authorization server stored under key `selected_AS` under $lsid$ in the client's session storage. As an honest client never changes this value once it is set, we know from (c) that the PAR request is sent to as. An honest authorization server – such as as – only reads a value stored under the key `code_challenge` in an incoming message when processing a request to its `/par` endpoint (Lines 53ff. of Algorithm 9). There, the value stored under `code_challenge` – i.e., $\mathsf{hash}(codeVerifier)$ – is stored in an authorization request record in the authorization server's authorization requests storage (see Lines 67, 77, and 78). Since as is honest, it never sends out the `code_challenge` value (neither from the authorization requests storage, nor from the records storage to which the `code_challenge` is copied in Line 44 of Algorithm 9). Hence, the value $\mathsf{hash}(codeVerifier)$ sent in the PAR request is not leaked "directly".
However, this value would be derivable if $codeVerifier$ leaks, i.e., we also have to prove that $codeVerifier$ does not leak. As noted above, this value is a fresh nonce stored in c's session storage under the key `code_verifier`. The only place in which a client accesses such a value is in function SEND_TOKEN_REQUEST, where the value is included in the body of an HTTPS request under the key `code_verifier` (Lines 6f. of Algorithm 3) which is sent to the token endpoint of the authorization server stored under key `selected_AS` under $lsid$ in the client's session storage – i.e., as by (c). Hence, this request in itself does not leak $codeVerifier$.
The only place in which an honest authorization server reads a value stored under the key `code_verifier` from an incoming message is when processing a token request in Line 89 of Algorithm 9. This value is not stored by the authorization server, neither is it sent anywhere. Hence, $codeVerifier$ does not leak.

(II) as **stores** $\mathsf{hash}(codeVerifier)$ **in** S**.** Because the cookie from (a) includes the `__Host` prefix and b is honest, that cookie must have been set by c. Clients only ever set such cookies when processing PAR responses in Lines 21ff. of Algorithm 2. With (b) (note that a client will never change the value stored under `code_verifier`), this implies that c sent a PAR request containing $\mathsf{hash}(codeVerifier)$ to as (see (I)) and got a response. Hence, as must have processed that PAR request as described above. Part of that processing is to store the value of `code_challenge` from the request – i.e., $\mathsf{hash}(codeVerifier)$ here – in the authorization request storage. Thus, we can conclude that there must be some $requestUri'$ such that $S(as).\texttt{authorizationRequests}[requestUri'][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$.

(III) **Proof for 1).** From (I), we have that only c and as know the value $\mathsf{hash}(codeVerifier)$ and do not use it in any request except for a single PAR request from c to as. From (II), we have that as stores $\mathsf{hash}(codeVerifier)$ as part of processing that PAR request. As as will use a fresh nonce as request URI for every processed PAR request (see Line 70 of Algorithm 9), and never changes the stored values (except for `code`), we can conclude that there is exactly one $requestUri$ such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$.

(IV) **Proof for 2).** As shown above, $requestUri$ is a fresh nonce chosen and stored by as when processing a PAR request send by c. $requestUri$ is not sent out by authorization servers anywhere, except in the response to the PAR request (under the key `request_uri`) that lead to the "creation" of $requestUri$.

Since we already established that the receiver of that PAR response is c (see above), we now have to check how c uses $requestUri$. c only reads a value stored under the key `request_uri` from an incoming message when processing the response to a PAR request. While c does store that value in its session storage, it never accesses that stored value. However, after processing the PAR response, c constructs an authorization request containing $requestUri$ as part of the query parameters (under key `request_uri`). That authorization request is a redirect which "points" to the authorization endpoint of the authorization server stored under key `selected_AS` under $lsid$ in c's session storage (i.e., as by (c)). By (d), we also know that c does not execute Line 36 of Algorithm 2, i.e., does not leak the authorization request for $lsid$. Before looking at the receiver of the aforementioned redirect, we note that as only ever reads the value of a request parameter `request_uri` in Line 19 of Algorithm 9 – that value is neither stored, nor sent out by as.

The redirect sent out by c when processing the PAR response is an HTTPS response which – among other things – contains a Set-Cookie header with a cookie of the form $\langle\langle \texttt{__Host}, \texttt{sessionId}\rangle, \langle lsid, \top, \top, \top\rangle\rangle$. We note that this is the only place in which c sets such a cookie.

Since we know from (a) that b knows such a cookie, and (II) implies that c must have set this cookie, we know that the HTTPS response containing the redirect with $requestUri$, sent by c, was sent to b.

We now only have to show that b does not leak $requestUri$. The aforementioned redirect contains a Location header (Line 29 of Algorithm 2) and status code 303, hence b will enter the location header handling in Line 11 of Algorithm 21 when processing that redirect (note that the redirect is sent by c with an empty script, i.e., no leakage through a script is possible). This handling will either end in a **stop** without any changes to b's state and no output event – which means that b does neither store, nor send out $requestUri$ – or with a call of HTTP_SEND in Line 27 of Algorithm 21. While HTTP_SEND does store the message to be send (containing $requestUri$), that stored value is only ever accessed when processing a DNS response and is then encrypted and sent out. We already established above that the redirection target is one of as's authorization endpoints and that as does not leak any $requestUri$ values received there. Hence, we have that only b, c, and as know $requestUri$, i.e., for all processes $p \notin \{b, c, as\}$, we have $requestUri \notin d_\emptyset(S(p))$. ■

B. Authorization Property

In this section, we show that the authorization property from Definition 15 holds.

Lemma 16 (Authorization). For

- every run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ of $\mathcal{FAPI}$ with a network attacker,
- every resource server $rs \in \mathsf{RS}$ that is honest in S^n,
- every identity $id \in^{\langle\rangle} s_0^{rs}.\texttt{ids}$ with $b = \mathsf{ownerOfID}(id)$ being an honest browser in S^n,
- every processing step in ρ

$$Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \to E^Q_{\text{out}}]{e^Q_{\text{in}} \to rs} (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$$

- every $resourceID \in \mathbb{S}$ with $as = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ being honest in S^Q,

it holds true that:

If $\exists x, y, k, m_{\text{resp}}.\langle x, y, \mathsf{enc_s}(m_{\text{resp}}, k)\rangle \in^{\langle\rangle} E^Q_{\text{out}}$ such that m_{resp} is an HTTP response with $m_{\text{resp}}.\texttt{body}[\texttt{resource}] \in^{\langle\rangle} S^{Q\prime}(rs).\texttt{resourceNonce}[id][resourceID]$, then

1) There exists a processing step

$$P = (S^P, E^P, N^P) \xrightarrow[rs \to E^P_{\text{out}}]{e^P_{\text{in}} \to rs} (S^{P\prime}, E^{P\prime}, N^{P\prime})$$

such that

a) either $P = Q$ or P prior to Q in ρ, and
b) e^P_{in} is an event $\langle x, y, \mathsf{enc_a}(\langle m_{\text{req}}, k_1\rangle, k_2)\rangle$ for some x, y, k_1, and k_2 where $m_{\text{req}} \in \mathcal{T}_{\mathcal{N}}$ is an HTTP request which contains a term (access token) t in its `Authorization` header, i.e., $t \equiv m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$, and
c) r was generated in P in Line 46 of Algorithm 12.

2) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, as, and id in S^Q (see Definition 14).
3) If there exists a client $c \in \mathsf{C}$ such that $k \equiv \mathsf{pub}(\mathsf{signkey}(c))$ or $k \equiv \mathsf{pub}(\mathsf{tlskey}(d_c))$ and $d_c \in \mathsf{dom}(c)$, and if c is honest in S^n, then r is not derivable from the attackers knowledge in S^n (i.e., $r \notin d_\emptyset(S^n(\mathsf{attacker}))$).

PROOF. **Resource server sends resource to correct client.** The first and the second postcondition are shown in Lemma 12, where we also showed that the message contained in the event e^P_{in} was created by c (as intuitively, the access token is bound to c via mTLS or DPoP, and no other process can prove possession of the secret key to which the token is bound). The resource r is sent back as a response to e^P_{in}: If the resource server sends out the resource in Line 69 of Algorithm 12, then it encrypts the HTTPS response (symmetrically) with the key contained in e^P_{in}. Otherwise, the resource server sends

out the response in Line 22 of Algorithm 13, encrypted (symmetrically) with the key contained in e^P_{in} (the resource server stored the key in its state).
Thus, the resource server sends out the resource r back to c, encrypted with a symmetric key that only c and rs can derive.

Client never sends resource r to attacker. In the following, we show that c does not send the resource nonce r to the attacker by contradiction, i.e., we assume that the client does send r to the attacker.

Redirection request was created by attacker. The client processes the response of the resource server (containing the resource r) in Line 49 of Algorithm 2 in some processing step $R = (S^R, E^R, N^R) \xrightarrow[c \to E^R_{\text{out}}]{e^R_{\text{in}} \to c} (S^{R\prime}, E^{R\prime}, N^{R\prime})$ (R happens after Q). The client stores the resource into its sessions in Line 51 of Algorithm 2, but never access it again in any other location. The client sends the resource as a response to a request req_{redir} stored in $S^R.\texttt{sessions}[sessionId[\texttt{redirectEpRequest}]$, for some value $sessionId$ (and in particular, encrypts the response with the key contained in req_{redir}), see Line 11, Line 53, and Line 54 of Algorithm 2.
An honest client sets `redirectEpRequest` values only in the redirection endpoint in Line 12 of Algorithm 1, i.e., req_{redir} is a request that was previously received by the client. This request contains a value (an authorization code) in $req_{\text{redir}}.\texttt{parameters}[\texttt{code}]$ (which the client puts into the token request in Algorithm 3).
As we assume that the client sends r to the attacker, it follows that req_{redir} was created by the attacker, in particular, the attacker can derive the symmetric key and all other values in the request.

Access token was sent by correct authorization server. Before sending the resource request, the client ensures that it sent the token request to the correct authorization server, i.e., the authorization server managing the resource: The client sends resource requests only in Algorithm 4. In Line 7 of Algorithm 4, the client checks whether the input argument $tokenEPDomain$ is a domain of the authorization server managing the resource that the client wants to request at the resource server. Algorithm 4 is called only in Line 44 of Algorithm 2, and the value $tokenEPDomain$ is the domain of the token request, i.e., the client received the access token from $\mathsf{authorizationServerOfResource}^{rs}(resourceID)$ (see Definition 11). This authorization server is honest, as required by the precondition of the lemma.

Attacker can derive authorization code issued for honest client and id. The access token that the client received in the token request is bound to its signing key or TLS key, the authorization server as, and the identity id (as shown in the second postcondition, it is bound to some key, as, and id, and the third postcondition has the requirement that the key is a key of the client). The authorization server created the access token in the token endpoint in Line 81 of Algorithm 9 in some processing step $T = (S^T, E^T, N^T) \xrightarrow[as \to E^T_{\text{out}}]{e^T_{\text{in}} \to as} (S^{T\prime}, E^{T\prime}, N^{T\prime})$. The token request contains a code $code$ such that there is a record $rec \in^{\langle\rangle} S^T(as).\texttt{records}$ with $rec[\texttt{code}] \equiv code$ and $code \not\equiv \bot$ (Line 92 of Algorithm 9). Furthermore, the record has the following values:

- $rec[\texttt{clientId}]$ is a client identifier of c at as, as the token request was sent by c
- $rec[\texttt{subject}] \equiv id$ (as the access token is bound to this identity)

As shown in Lemma 3, the code that the client uses is the same code that it received in the request to the redirection endpoint, i.e., req_{redir}.
However, this is a contradiction to Lemma 14, i.e., such an authorization code cannot leak to the attacker.

■

C. Authentication Property

In this section, we show that the authentication property from Definition 17 holds. This will be a proof by contradiction, i.e., we assume that there is a FAPI web system $\mathcal{FAPI}$ in which the authentication property is violated and deduce a contradiction.

Assumption 1. There exists a FAPI web system with a network attacker $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ with a configuration (S, E, N) in ρ, some $c \in \mathsf{C}$ that is honest in S, some identity $id \in \mathsf{ID}$ with $as = \mathsf{governor}(id)$ being an honest AS and $b = \mathsf{secretOfID}(id)$ being browser honest in S, some service session identified by some nonce n for id at c, and n is derivable from the attacker's knowledge in S (i.e., $n \in d_\emptyset(S(\mathsf{attacker}))$).

Lemma 17 (Authentication Property Holds). Assumption 1 is a contradiction.

PROOF. By Assumption 1, there is a service session identified by n for id at c, and hence, by Definition 16, we have that there is a session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ with $S(c).\texttt{sessions}[x][\texttt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(c).\texttt{sessions}[x][\texttt{serviceSessionId}] \equiv n$. Assumption 1 says that n is derivable from the attacker's knowledge. Since we have $S(c).\texttt{sessions}[x][\texttt{serviceSessionId}] \equiv n$, we can check where such an entry in c's state can be created.

The only place in which an honest client stores a service session id is in the function CHECK_ID_TOKEN, specifically in Line 15 of Algorithm 5. There, the client chooses a fresh nonce as the value for the service session id, in this case n. In the line before, it sets the value for $S(c).\texttt{sessions}[x][\texttt{loggedInAs}]$, in this case $\langle d, id\rangle$.

CHECK_ID_TOKEN, in turn, is only called in a single place: When processing an HTTPS response to a token request, in Line 48 of Algorithm 2. From the check in Line 47 of Algorithm 2, we know that this response came from as's token endpoint. Since as is an honest authorization server, it will only reply to a token request if that request contains a valid authorization code, i.e., the token request must contain a $code$ such that there is a record $rec \in^{\langle\rangle} S(as).\texttt{records}$ with $rec[\texttt{code}] \equiv code$, $rec[\texttt{client\_id}] \equiv clientId$, and $rec[\texttt{subject}] \equiv id$ where $clientId$ must be one of c's identifiers at as (otherwise, client authentication would fail and as would not output an id token, see Lemma 8).

By tracking backwards from Line 15 of Algorithm 5, it is easy to see that the same party that finally receives the service session id n in an HTTPS response sent in Line 19 of Algorithm 5 must have sent an HTTPS request req to c containing the aforementioned $code$.

We now have to differentiate between two cases: Either (a) the sender of req is one of b, c, as; or (b) the sender of req is any other process (except for b, c, and as).

In case (a), we know that the only party sending an HTTPS request with an authorization code (i.e., with a body dictionary containing a key `code`) is b. If b sent req, b receives the service session id n in a set-cookie header with the httpOnly and secure flags set (see Line 17 of Algorithm 5). Hence, b will only ever send n to c in a cookie header as part of HTTPS requests, which does not leak n. Neither does c leak received service session id cookie values – in fact, c never even accesses a cookie named `serviceSessionId`. Furthermore, neither b, nor c leak n in any other way (the value is not even accessed), resulting in a contradiction to Assumption 1.

In case (b), that other process which sent req would need to know $code$ in order to be able to include it in req. This contradicts Lemma 14. ■

D. Session Integrity for Authentication Property

In this section, we show that the authentication property from Definition 23 holds.

Assumption 2. There exists a FAPI web system with a network attacker $\mathcal{FAPI}$ such that there exists a run ρ of $\mathcal{FAPI}$ with a processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, a browser b honest in S, an authorization server $as \in \mathsf{AS}$, an identity id, a client $c \in \mathsf{C}$ honest in S, and a nonce $lsid$ with $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$ and c did not leak the authorization request for $lsid$, such that

(1) there is no processing step Q' prior to Q in ρ such that $\mathsf{started}^{Q'}_{\rho}(b, c, lsid)$, or

(2) as is honest in S, and there is no processing step Q'' prior to Q in ρ such that $\mathsf{authenticated}^{Q''}_{\rho}(b, c, id, as, lsid)$.

Lemma 18 (Session Integrity for Authentication Property Holds). Assumption 2 is a contradiction.

PROOF. **(1).** We have that $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$. With Definition 18, we know that c sent out a service session id associated with $lsid$ to b. This can only happen when the client's function CHECK_ID_TOKEN was called with $lsid$ as the first argument – which, in turn, can only happen in Line 48 of Algorithm 2 when c processes a response to a token request. Such a response is only accepted by c if c sent a corresponding token request before. Clients only send token requests in Line 21 of Algorithm 1, when processing an HTTPS request req_{redir}.

req_{redir} must contain a cookie $[\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle : lsid]$: The response (containing the service session id) sent by c to b in Line 19 of Algorithm 5 is sent to and encrypted for the sender of req_{redir}, because c looks these values up in the login session record stored in $S(c).\texttt{sessions}[lsid]$ under the key `redirectEpRequest`. Such an entry in c's session storage is only ever created in Line 20 of Algorithm 1 when processing an HTTPS request containing a cookie as described above.

We can now track how that cookie was stored in b: Since the cookie is stored under a domain of c (otherwise, b would not include it in requests to c) and the cookie is set with the `__Host` prefix, the cookie must have been set by c. A cookie with the properties shown above is only set in Line 30 of Algorithm 2. Similar to the `redirectEpRequest` session entry above, c sends this cookie as a response to a stored request, in this case, using the key `startRequest` to determine receiver and encryption key. A session entry with key `startRequest` is only ever created in Line 10 of Algorithm 1. Hence, for b to receive the cookie, there must have been a request from b to c to the `/startLogin` endpoint, using the POST method, and with an origin header for an origin of c (see Line 6 of Algorithm 1).

Due to the origin check, this request must have been sent by a script under one of c's origins. There is only one script which could potentially send such a request: $script\_client\_index$. Hence, there must be a processing step Q' (prior to Q) in ρ in which b executed $script\_client\_index$ and in that script, executed Line 8 of Algorithm 8.

In addition, we already established above that c replied to this request (stored under the key `startRequest`) with a response containing a header of the form $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle : \langle lsid, \top, \top, \top\rangle]\rangle$.

Hence, we have that $\mathsf{started}^{Q'}_{\rho}(b, c, lsid)$.

(2). Again, we have $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ and we know that c sent out a service session id associated with $lsid$ to b. This can only happen in the client's function CHECK_ID_TOKEN, which only produces an output if c received an id token t (via a token response). From $S(c).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$, we know that for $t_c := \mathsf{extractmsg}(t)$, we have $t_c[\texttt{iss}] \equiv d$, $t_c[\texttt{sub}] \equiv id$, and $t_c[\texttt{aud}] \equiv clientId$ (for some $clientId$). Due to the check in Line 47 of Algorithm 2, this id token must have been sent by as (because $d \in \mathsf{dom}(as)$). as will only output such a term t if there is a record rec in as's `records` state subterm with $rec[\texttt{subject}] \equiv id$, $rec[\texttt{client\_id}] \equiv clientId$, $rec[\texttt{code\_challenge}] \equiv codeChallenge$ (for some value of $codeChallenge$). We note that as issuing an id token with $t_c[\texttt{aud}] \equiv clientId$ implies that c has client identifier $clientId$ at as (see Definition 8 and Definition 12).

By construction of c and tracking of $\texttt{sessions}[lsid]$ in c's state, it is easy to see that once c reaches CHECK_ID_TOKEN, the session storage $S(c).\texttt{sessions}[lsid]$ must contain a key `code_verifier` under which a nonce $codeVerifier$ is stored. We note that $S(b).\texttt{cookies}[d_c]$ must contain a cookie $\langle\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle, \langle lsid, \top, \top, \top\rangle\rangle$ for $d_c \in \mathsf{dom}(c)$, because b sends a cookie $[\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle : lsid]$ as explained above, b is honest (and will thus not accept `__Host` headers for d_c from parties other than c), and if c sets a cookie $\langle\texttt{\_\_Host}, \texttt{sessionId}\rangle$, it will do with the attributes set as shown here.

Hence, we can apply Lemma 15 (note that $S(c).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$ with $d \in \mathsf{dom}(as)$ implies $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}] \equiv d \in \mathsf{dom}(as)$). I.e., we now have that there is exactly one $requestUri$ such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$, and only b, c, and as know $requestUri$.

We know from Line 95 of Algorithm 9 that the token request which leads to as issuing t must contain a code verifier such that $\mathsf{hash}(codeVerifier) \equiv rec[\texttt{code\_challenge}]$ (with rec from above). Since we know that c must have sent the token request (otherwise, c would not have received t), we can track where and how c creates such a request. This is only the case in function SEND_TOKEN_REQUEST. There, c selects the value for the code verifier and based on the session id which c received from b via the `sessionId` cookie. At the same time, c includes the $code$ from b's request's parameters.

Going back to as, we can track where a rec as described above can be stored into as's state: This is only the case at as's `/auth2` endpoint (Lines 31ff. of Algorithm 9). There, as will only store a record rec, if there is an $authZrec$, stored under the key $reqUri$ in the `authorizationRequests` state subterm such that there is an $auth2Reference$ with $authZrec[\texttt{auth2\_reference}] \equiv auth2Reference$ and that $auth2Reference$ is contained in the request to as's `/auth2` endpoint. Such an $auth2Reference$, in turn, is only created at as's `/auth` endpoint. For a request to this endpoint to lead to storing $auth2Reference$, the request must contain $reqUri$ under the key `request_uri`.

Note that by Lemma 15, we established that there is exactly one $requestUri$ in as's state such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{code\_challenge}] \equiv \mathsf{hash}(codeVerifier)$. Therefore, $reqUri \equiv requestUri$. In addition, it is easy to see that c and as do not send any requests to as's `/auth` endpoint. Hence, b must have sent a request with $reqUri$ to `/auth`.

Since $auth2Reference$ from above is only sent to whoever sent the first request to `/auth` (and – if b receives it – b does not leak that value) we know that b must have sent the request to `/auth2` as well. As b is honest, this can only happen trough a script – together with the origin header check in Line 31 of Algorithm 9, and $script\_as\_form$ being the only script ever sent by as, we can conclude that there must have been a processing step Q'' prior to Q' in ρ in which b was triggered, selected a document under one of as's origins with script $script\_as\_form$, executed that script, selected id from its identities (because we know from above that $rec[\texttt{subject}] \equiv id$ and such a rec is only stored at `/auth2` endpoint, if the identity in the request is equivalent to id) and sent a request to as's `/auth2` endpoint containing $auth2Reference$ – hence, the $scriptstate$ contained a key `auth2_reference` with value $auth2Reference$.

Hence, we have $\mathsf{authenticated}_\rho^{Q''}(b, c, id, as, lsid)$ which – together with (1). from above – contradicts Assumption 2, therefore proving the lemma.

■

E. Session Integrity for Authorization Property

In this section, we show that the session integrity property from Definition 24 holds.

Lemma 19 (Session Integrity for Authorization Property Holds). For every run ρ of $\mathcal{FAPI}$, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ, 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 $\mathsf{accessesResource}_\rho^{\mathsf{Q}}(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 22), 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 $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.

PROOF. **(1).** Due to $\mathsf{accessesResource}_\rho^{\mathsf{Q}}(b, r, u, c, rs, as, lsid)$, it holds true that the browser b has a `sessionid` cookie with the session identifier $lsid$ for the domain of the client c. This cookie is set with the `__Host` prefix, i.e., it follows that the cookie was set by c, which responds with a $\texttt{Set} - \texttt{Cookie}$ (with `sessionid`) only in Line 30 of Algorithm 2. The remaining

proof is analogous to the proof of the first postcondition of Lemma 18.

(2).

Client received resource from rs. As the client executes Line 55 of Algorithm 2 (precondition of the lemma), and as $S'(c).\texttt{sessions}[lsid][\texttt{resourceServer}] \in \texttt{dom}(rs)$ is only set in Line 52 of Algorithm 2, it follows that c received the resource r in a response from rs, i.e., it sent the corresponding resource request to rs.

Resource request contains access token associated with u at as. An honest resource server sends out an HTTP response $resp_{\text{resource}}$ with $\texttt{resource} \in^{\langle\rangle} resp_{\text{resource}}.\texttt{body}$ either in Line 69 of Algorithm 12 or Line 22 of Algorithm 13. As shown in the proof of Lemma 16, the resource server received a resource request req_{resource} containing an access token t (either in the same processing step when storing the resource or in a previous processing step). Furthermore, as the resource server stores the resource in $S(rs).\texttt{resourceNonces}[u][resourceId]$ (with $resourceId \in \mathcal{T}_{\mathcal{N}}$), it follows that req_{resource} has the path $resourceId$. Thus, it follows that the value $responsibleAS$ chosen by the resource server in Line 15 of Algorithm 12 is a domain of as (as the resource server never changes the `resourceASMapping` subterm of its state, see also Definition 13).

If rs returns the resource r in Line 69 of Algorithm 12, then the access token is a structured JWT signed by as (Line 60 of Algorithm 12) and containing the `sub` value u (Line 62 of Algorithm 12). Otherwise, if r is returned in Line 22 of Algorithm 13, then the resource server received a response from as containing the `sub` value u and asserting that the access token contained in req_{resource} is valid. In both cases (structured access token or opaque token) it follows that the authorization server as has a sequence rec in the `records` subterm of its state with $rec[\texttt{access\_token}] \equiv t$ and $rec[\texttt{subject}] \equiv u$.

Token request was sent to as. An honest client sends resource requests only in Algorithm 4, which is called only in Line 44 of Algorithm 2, i.e., after receiving the token response. The check in Line 7 of Algorithm 4 ensures that the token request req_{token} was sent to as (as the client calls Algorithm 4 with the domain of the token request, see also Definition 11). From this, it follows that $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}]$ is a domain of as, as the client sends the token request to this domain, see Line 4, Line 10, and Line 11 of Algorithm 3.

PAR request was sent to as. The token request req_{token} sent from c to as contains an authorization code $code$ and a PKCE code verifier $pkce\_cv$ (see Line 6 of Algorithm 3). As the authorization server responds with an access token, it follows that the checks at the token endpoint in Line 81 of Algorithm 9 passed successfully. In particular, this implies that the token request contains the correct PKCE verifier for the code, i.e., the authorization code and the PKCE challenge corresponding to the PKCE verifier were stored in the same record entry in the `records` state subterm (see Line 89 and Line 95 of Algorithm 9).

An authorization server adds these records to its `records` state subterm only in Line 44 of Algorithm 9, and the sequence that is added is taken from the `authorizationRequests` state subterm, see Line 40 of Algorithm 9. In this processing step, the authorization server also creates the authorization code (Line 43 of Algorithm 9) and associates the identity with the code (Line 41 of Algorithm 9).

Thus, as the authorization server as exchanged the authorization code $code$ at the token endpoint and the issued access token is associated with the identity u, it follows that identity u logged in at the `/auth2` endpoint of as, and the request to `/auth2` contained a value `auth2_reference` in its body equal to $S''(as).\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}]$ (with S'' being the state of a configuration prior to Q; see also Line 39 of Algorithm 9). The authorization server received the $requestUri$ value at the `auth` endpoint, i.e., the process that can derive the request URI value can also derive the `auth2` reference.

As $S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}]$ is a domain of as, it follows that the client sent the pushed authorization request to as in Line 36 of Algorithm 6 in a previous processing step of the trace. In this processing step, the client chose the PKCE verifier $pkce\_cv$ in Line 26 of Algorithm 6 and stored this value into the $lsid$ session in Line 34 of Algorithm 6. Now, we can apply Lemma 15 and conclude that the request URI can only be derived by b, c, and as. As as does not send requests to itself and c does not send any request to an $auth$ endpoint, it follows that the request to the $auth$ endpoint of as was sent by b. The remaining argumentation is the same as for the proof of Lemma 18.

■

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, 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.

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

[21] 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.

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

[23] 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.

[24] N. Sakimura. *FAPI WG Issue 543 – Browser swap attack explained on 2022-09-28*. 2022. URL: https://bitbucket.org/openid/fapi/issues/543/browser-swap-attack-explained-on-2022-09.

[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 25 (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 5). 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 26 (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 27 (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 28 (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 5. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 29 (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 $t \dot{\sim} \mathit{patterns}$ 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 30 (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 $i \in \{1, \dots, n\}$.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{5}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{6}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{7}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{8}$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \tag{9}$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \tag{10}$$
$$\pi_i(\langle x_1, \dots, x_n\rangle) = x_i \ \text{ if } 1 \leq i \leq n \tag{11}$$
$$\pi_j(\langle x_1, \dots, x_n\rangle) = \diamond \ \text{ if } j \notin \{1, \dots, n\} \tag{12}$$
$$\pi_j(t) = \diamond \ \text{ if } t \text{ is not a sequence} \tag{13}$$

Figure 5. Equational theory for Σ.

Definition 31 (Sequence Notations). Let $t = \langle t_1, \ldots, t_n \rangle$ and $r = \langle r_1, \ldots, r_m \rangle$ be sequences, s a set, and x, y terms. We define the following operations:

- $t \subset^{\langle\rangle} s \iff t_1, \ldots, t_n \in s$
- $x \in^{\langle\rangle} t \iff \exists i\colon\ t_i = x$
- $t +^{\langle\rangle} y := \langle t_1, \ldots, t_n, y \rangle$
- $t \cup r := \langle t_1, \ldots, t_n, r_1, \ldots, r_m \rangle$
- $t -^{\langle\rangle} y := \begin{cases} \langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \rangle & \text{if } \exists i\colon t_i = x \text{ (i.e., } y \in^{\langle\rangle} t\text{)} \\ t & \text{otherwise (i.e., } y \notin^{\langle\rangle} t\text{)} \end{cases}$
  If y occurs more than once in t, $t -^{\langle\rangle} y$ non-deterministically removes one of the occurrences.
- $t -^{\langle\rangle *} y$ is t with all occurrences of y removed.
- $|t| := n$. If t' is not a sequence, we set $|t'| := \diamond$.
- For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. The order of the elements does not matter; one is chosen arbitrarily.

Definition 32 (Dictionaries). A *dictionary over X and Y* is a term of the form

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

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1\colon v_1, \ldots, k_n\colon v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$. Note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$; and dictionaries as such may contain duplicate keys (however, all dictionary operations are only defined on dictionaries with unique keys).

Definition 33 (Operations on Dictionaries). Let $z = [k_1\colon v_1, k_2\colon v_2, \ldots, k_n\colon v_n]$ be a dictionary with unique keys, i.e., $\forall i, j\colon k_i \neq k_j$. In addition, let t and v be terms. We define the following operations:

- $t \in z \iff \exists i \in \{1, \ldots, n\}\colon k_i = t$
- $z[t] := \begin{cases} v_i & \text{if } \exists k_i \in z\colon t = k_i \\ \langle\rangle & \text{otherwise (i.e., if } t \notin z\text{)} \end{cases}$
- $z - t := \begin{cases} [k_1\colon v_1, \ldots, k_{i-1}\colon v_{i-1}, k_{i+1}\colon v_{i+1}, \ldots, k_n\colon v_n] & \text{if } \exists k_i \in z\colon t = k_i \\ z & \text{otherwise (i.e., if } t \notin z\text{)} \end{cases}$
- In our algorithm descriptions, we often write **let** $z[t] := v$. If $t \notin z$ prior to this operation, an element $\langle t, v \rangle$ is appended to z. Otherwise, i.e., if there already is an element $\langle t, x \rangle \in^{\langle\rangle} z$, this element is updated to $\langle t, v \rangle$.

We emphasize that these operations are only defined on dictionaries with unique keys.

Given a term $t = \langle t_1, \ldots, t_n \rangle$, 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 34 (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 $\langle host, protocol \rangle$ and o is an Origin term, then we can write $o.\texttt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Definition 35 (Concatenation of Sequences). For a sequence $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = \langle b_1, b_2, \ldots \rangle$, we define the *concatenation* as $a \cdot b := \langle a_1, \ldots, a_i, b_1, b_2, \ldots \rangle$.

Definition 36 (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 37. A *URL* is a term of the form

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

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

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

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 34):

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 38. 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 39. 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., $name.\texttt{1}$) 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.[8] 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 $name.\texttt{1} \equiv \texttt{\_\_Host}$), 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 40. An *HTTP request* is a term of the form shown in (14). An *HTTP response* is a term of the form shown in (15).

$$\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body \rangle \tag{14}$$

$$\langle \texttt{HTTPResp}, nonce, status, headers, body \rangle \tag{15}$$

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,

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

- $\langle \texttt{Cookie}, c \rangle$ where $c \in [\mathbb{S} \times \mathcal{T}_\mathcal{N}]$ (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 [20]),
- $\langle \texttt{ReferrerPolicy}, p \rangle$ where $p \in \{\texttt{noreferrer}, \texttt{origin}\}$.

- $body \in \mathcal{T}_\mathcal{N}$ in requests and responses.

We write HTTPRequests/HTTPResponses for the set of all HTTP requests or responses, respectively.

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, \texttt{1} \rangle\rangle, [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S} \rangle], \langle \texttt{foo}, \texttt{bar} \rangle \rangle \tag{16}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top \rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x \rangle \rangle \tag{17}$$

An HTTP `POST` request for the URL http://example.com/show?index=1 is shown in (16), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar} \rangle$. A possible response is shown in (17), 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 41. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k' \rangle, k)$, where $k \in terms$, $k' \in \mathcal{N}$, 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 HTTPSRequests or 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.

$$\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{18}$$

$$\mathsf{enc_s}(s, k') \tag{19}$$

The term (18) shows an encrypted request (with r as in (16)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (19) is a response (with s as in (17)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (18).

5) DNS Messages:

Definition 42. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, domain, nonce \rangle$ where $domain \in \mathsf{Doms}$, $nonce \in \mathcal{N}$. We call the set of all DNS requests DNSRequests.

Definition 43. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, domain, result, nonce \rangle$ with $domain \in \mathsf{Doms}$, $result \in \mathsf{IPs}$, $nonce \in \mathcal{N}$. We call the set of all DNS responses 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 44 (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 45 (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[9] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

Definition 46 (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

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) $E^\nu_{\text{out}} \subseteq 2^{\mathcal{T}^\nu \langle\rangle}$ of protoevents,
2) a term $s^\nu \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3) a sequence $(v_1, v_2, \dots, v_i)$ of all placeholders appearing in E^ν_{out} (ordered lexicographically),
4) a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ 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) $E_{\text{out}} = E^\nu_{\text{out}}[\eta_1/v_1, \dots, \eta_i/v_i]$,
3) $S'(p) = s^\nu[\eta_1/v_1, \dots, \eta_i/v_i]$ 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 47 (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 $((S^0, E^0, N^0), \dots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \dots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ 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 48 (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 $n \geq 0$, $m_1, \dots, m_n \in M$, and $\tau \in \mathcal{T}_\emptyset(\{x_1, \dots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \dots, m_n/x_n]$. 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 49 (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 $E, s' \in d_{V_{\text{process}}}(\{e, s\})$.

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

Placeholder	Usage
ν_1	Algorithm 22, new window nonces
ν_2	Algorithm 22, new HTTP request nonce
ν_3	Algorithm 22, lookup key for pending HTTP requests entry
ν_4	Algorithm 20, new HTTP request nonce (multiple lines)
ν_5	Algorithm 20, new subwindow nonce
ν_6	Algorithm 21, new HTTP request nonce
ν_7	Algorithm 21, new document nonce
ν_8	Algorithm 17, lookup key for pending DNS entry
ν_9	Algorithm 14, new window nonce
$\nu_{10}, \ldots$	Algorithm 20, replacement for placeholders in script output

Table II: List of placeholders used in browser algorithms.

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 50 (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 $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_1, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$.

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 `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 FUNCTION_NAME(x, y)

to describe that a function 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 $z :=$ 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 25). 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: $\langle \texttt{URL}, \texttt{S}, d, path, \langle\rangle\rangle$. 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 51 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 52 (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 53 (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 54. A *window* is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$ Documents (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ if $documents$ is not empty (we then call d the *active document of* w). We write Windows for the set of all windows. We write $w.\texttt{activedocument}$ 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., $b.\texttt{opener} = a.\texttt{nonce}$.

Definition 55. A *document* d is a term of the form

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

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

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

Definition 56. For two window terms w and w' we write

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

if $w \in^{\langle\rangle} w'.\texttt{activedocument.subwindows}$. 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 57. A reference for a normal HTTP(S) request is a sequence of the form $\langle \texttt{REQ}, nonce \rangle$, where $nonce$ is a window reference. A reference for a XMLHttpRequest is a sequence of the form $\langle \texttt{XHR}, nonce, xhrreference \rangle$, where $nonce$ is a document reference and $xhrreference$ 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 32.

Definition 58. The *set of states* $Z_{\text{webbrowser}}$ *of a web browser atomic Dolev-Yao process* consists of the terms of the form

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

with the subterms as follows:

- $windows \subset^{\langle\rangle}$ 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 $OR := \{\langle o, r\rangle \mid o \in \mathsf{Origins}, r \in \mathcal{N}\}$ 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.
- $sts \subset^{\langle\rangle}$ Doms stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in$ 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.
- $isCorrupted \in \{\perp, \texttt{FULLCORRUPT}, \texttt{CLOSECORRUPT}\}$ 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[10] to windows in the window list $s.\texttt{windows}$ 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 59. 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 $w \in^{\langle\rangle} s.\texttt{windows}$ 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 $s.\overline{p}.\texttt{activedocument} \not\equiv \langle\rangle$, there exists a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

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 60. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\texttt{windows}$ 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 `headers` for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

[10]Recall the definition of a pointer in Definition 34.

Algorithm 14 Web Browser Model: Determine window for navigation.

```
1: function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s′)
2:     if window ≡ _BLANK then    → Open a new window when _BLANK is used
3:         if noreferrer ≡ ⊤ then
4:             let w′ := ⟨ν9, ⟨⟩, ⊥⟩
5:         else
6:             let w′ := ⟨ν9, ⟨⟩, s′.w̄.nonce⟩
7:         let s′.windows := s′.windows +^⟨⟩ w′
             ↪ and let w̄′ be a pointer to this new element in s′
8:         return w̄′
9:     let w̄′ ← NavigableWindows(w̄, s′) such that s′.w̄′.nonce ≡ window
         ↪ if possible; otherwise return w̄
10:    return w̄′
```

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 61. For a sequence of cookies (with pairwise different names) *oldcookies*, a sequence of cookies *newcookies*, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$, the set CookieMerge(*oldcookies*, *newcookies*, *protocol*) is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$ or where $(c.\texttt{name}.1 \equiv \texttt{\_\_Host}) \wedge ((protocol \equiv \texttt{P}) \vee (c.\texttt{secure} \equiv \bot))$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\texttt{name} \equiv c'.\texttt{name}$, 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*, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of 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 62. For a sequence of cookies (with pairwise different names) *oldcookies*, a cookie c, and a string $protocol \in \{\texttt{P}, \texttt{S}\}$ (denoting whether the HTTP response was received from an insecure or a secure origin), the sequence AddCookie(*oldcookies*, c, *protocol*) is defined by the following algorithm: Let $m :=$ *oldcookies*. If $(c.\texttt{name}.1 \equiv \texttt{\_\_Host}) \wedge \neg((protocol \equiv \texttt{S}) \wedge (c.\texttt{secure} \equiv \top))$, then return m, else: Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. 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 63. The set NavigableWindows($\overline{w}, s'$) is the set $\overline{W} \subseteq$ 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 $\overline{w'} \in$ Subwindows(s') the following is true:

- If $s'.\overline{w'}.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists\, \overline{w''} \in$ 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 $\exists\, \overline{p} \in$ Subwindows(s') such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\ s'.\overline{p}.\texttt{activedocument.origin} = s'.\overline{w}.\texttt{activedocument.origin}$ ($\overline{w'}$ 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 $\exists\, \overline{p} \in$ Subwindows(s') such that $s'.\overline{w'}.\texttt{opener} = s'.\overline{p}.\texttt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ 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 14) 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 it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':

Algorithm 15 Web Browser Model: Determine same-origin window.

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

Algorithm 16 Web Browser Model: Cancel pending requests for given window.

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

Algorithm 17 Web Browser Model: Prepare headers, do DNS resolution, save message.

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

Algorithm 18 Web Browser Model: Navigate a window backward.

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

Algorithm 19 Web Browser Model: Navigate a window forward.

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

Algorithm 20 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT(w̄, d̄, a, s′)
2:     let tree := Clean(s′, s′.d̄)
3:     let cookies := ⟨{⟨c.name, c.content.value⟩ | c ∈^⟨⟩ s′.cookies [s′.d̄.origin.host]
       ↪  ∧ c.content.httpOnly ≡ ⊥
       ↪  ∧ (c.content.secure ≡ ⊤ ⟹ (s′.d̄.origin.protocol ≡ S))}⟩
4:     let tlw ← s′.windows such that tlw is the top-level window containing d̄
5:     let sessionStorage := s′.sessionStorage [⟨s′.d̄.origin, tlw.nonce⟩]
6:     let localStorage := s′.localStorage [s′.d̄.origin]
7:     let secrets := s′.secrets [s′.d̄.origin]
8:     let R := script^−1(s′.d̄.script) if possible; otherwise stop
9:     let in := ⟨tree, s′.d̄.nonce, s′.d̄.scriptstate, s′.d̄.scriptinputs, cookies,
       ↪  localStorage, sessionStorage, s′.ids, secrets⟩
10:    let state′ ← T_N(V_process), cookies′ ← Cookies^ν, localStorage′ ← T_N(V_process),
       ↪  sessionStorage′ ← T_N(V_process), command ← T_N(V_process),
       ↪  out := ⟨state′, cookies′, localStorage′, sessionStorage′, command⟩
       ↪  such that out := out^λ[ν10/λ1, ν11/λ2, ...] with (in, out^λ) ∈ R
11:    let s′.cookies [s′.d̄.origin.host] :=
       ↪  ⟨CookieMerge(s′.cookies [s′.d̄.origin.host], cookies′, s′.d̄.origin.protocol)⟩
12:    let s′.localStorage [s′.d̄.origin] := localStorage′
13:    let s′.sessionStorage [⟨s′.d̄.origin, tlw.nonce⟩] := sessionStorage′
14:    let s′.d̄.scriptstate := state′
15:    let referrer := s′.d̄.location
16:    let referrerPolicy := s′.d̄.headers[ReferrerPolicy]
17:    let docorigin := s′.d̄.origin
18:    switch command do
19:        case ⟨HREF, url, hrefwindow, noreferrer⟩
20:            let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, noreferrer, s′)
21:            let reference := ⟨REQ, s′.w̄′.nonce⟩
22:            let req := ⟨HTTPReq, ν4, GET, url.host, url.path, url.parameters, ⟨⟩, ⟨⟩⟩
23:            if noreferrer ≡ ⊤ then
24:                let referrerPolicy := noreferrer
25:            let s′ := CANCELNAV(reference, s′)
26:            call HTTP_SEND(reference, req, url, ⊥, referrer, referrerPolicy, a, s′)
27:        case ⟨IFRAME, url, window⟩
28:            if window ≡ _SELF then
29:                let w̄′ := w̄
30:            else
31:                let w̄′ := GETWINDOW(w̄, window, s′)
32:            let req := ⟨HTTPReq, ν4, GET, url.host, url.path, url.parameters, ⟨⟩, ⟨⟩⟩
33:            let w′ := ⟨ν5, ⟨⟩, ⊥⟩
34:            let s′.w̄′.activedocument.subwindows := s′.w̄′.activedocument.subwindows +^⟨⟩ w′
35:            call HTTP_SEND(⟨REQ, ν5⟩, req, url, ⊥, referrer, referrerPolicy, a, s′)
```

  - If *window* is the string _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 NavigableWindows above).

  In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).
- The function GETWINDOW (Algorithm 15) 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 CANCELNAV (Algorithm 16) 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 HTTP_SEND (Algorithm 17) takes an HTTP request *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'.pendingDNS until the DNS resolution finishes. *reference* is a reference as defined in Definition 57. *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 18) and NAVFORWARD (Algorithm 19), navigate a window backward or forward.

```
36:     case ⟨FORM, url, method, data, hrefwindow⟩
37:         if method ∉ {GET, POST} then
38:             stop
39:         let w' := GETNAVIGABLEWINDOW(w, hrefwindow, ⊥, s')
40:         let reference := ⟨REQ, s'.w'.nonce⟩
41:         if method = GET then
42:             let body := ⟨⟩
43:             let parameters := data
44:             let origin := ⊥
45:         else
46:             let body := data
47:             let parameters := url.parameters
48:             let origin := docorigin
49:         let req := ⟨HTTPReq, ν4, method, url.host, url.path, parameters, ⟨⟩, body⟩
50:         let s' := CANCELNAV(reference, s')
51:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s')
52:     case ⟨SETSCRIPT, window, script⟩
53:         let w' := GETWINDOW(w, window, s')
54:         let s'.w'.activedocument.script := script
55:         stop ⟨⟩, s'
56:     case ⟨SETSCRIPTSTATE, window, scriptstate⟩
57:         let w' := GETWINDOW(w, window, s')
58:         let s'.w'.activedocument.scriptstate := scriptstate
59:         stop ⟨⟩, s'
60:     case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
61:         if method ∈ {CONNECT, TRACE, TRACK} ∨ xhrreference ∉ V_process ∪ {⊥} then
62:             stop
63:         if url.host ≢ docorigin.host ∨ url.protocol ≢ docorigin.protocol then
64:             stop
65:         if method ∈ {GET, HEAD} then
66:             let data := ⟨⟩
67:             let origin := ⊥
68:         else
69:             let origin := docorigin
70:         let req := ⟨HTTPReq, ν4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
71:         let reference := ⟨XHR, s'.d.nonce, xhrreference⟩
72:         call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s')
73:     case ⟨BACK, window⟩
74:         let w' := GETNAVIGABLEWINDOW(w, window, ⊥, s')
75:         call NAVBACK(w', s')
76:     case ⟨FORWARD, window⟩
77:         let w' := GETNAVIGABLEWINDOW(w, window, ⊥, s')
78:         call NAVFORWARD(w', s')
79:     case ⟨CLOSE, window⟩
80:         let w' := GETNAVIGABLEWINDOW(w, window, ⊥, s')
81:         remove s'.w' from the sequence containing it
82:         stop ⟨⟩, s'
83:     case ⟨POSTMESSAGE, window, message, origin⟩
84:         let w' ← Subwindows(s') such that s'.w'.nonce ≡ window
85:         if ∃j ∈ ℕ such that s'.w'.documents.j.active ≡ ⊤
              ↪ ∧ (origin ≢ ⊥ ⟹ s'.w'.documents.j.origin ≡ origin) then
86:             let s'.w'.documents.j.scriptinputs := s'.w'.documents.j.scriptinputs
                  ↪ +^⟨⟩ ⟨POSTMESSAGE, s'.w.nonce, docorigin, message⟩
87:         stop ⟨⟩, s'
88:     case else
89:         stop
```

Algorithm 21 Web Browser Model: Process an HTTP response.

1: **function** PROCESSRESPONSE(*response*, *reference*, *request*, *requestUrl*, *a*, *f*, *s′*)
2: **if** `Set-Cookie` ∈ *response*.`headers` **then**
3: **for each** *c* ∈⟨⟩ *response*.`headers`[`Set-Cookie`], *c* ∈ Cookies **do**
4: **let** *s′*.`cookies`[*request*.`host`]
↪ := AddCookie(*s′*.`cookies`[*request*.`host`], *c*, *requestUrl*.`protocol`)
5: **if** `Strict-Transport-Security` ∈ *response*.`headers` ∧ *requestUrl*.`protocol` ≡ `S` **then**
6: **let** *s′*.`sts` := *s′*.`sts` +⟨⟩ *request*.`host`
7: **if** `Referer` ∈ *request*.`headers` **then**
8: **let** *referrer* := *request*.`headers`[`Referer`]
9: **else**
10: **let** *referrer* := ⊥
11: **if** `Location` ∈ *response*.`headers` ∧ *response*.`status` ∈ {303, 307} **then**
12: **let** *url* := *response*.`headers`[`Location`]
13: **if** *url*.`fragment` ≡ ⊥ **then**
14: **let** *url*.`fragment` := *requestUrl*.`fragment`
15: **let** *method′* := *request*.`method`
16: **let** *body′* := *request*.`body`
17: **if** `Origin` ∈ *request*.`headers`
↪ ∧ *request*.`headers`[`Origin`] ≠ ◇
↪ ∧ (⟨*url*.`host`, *url*.`protocol`⟩ ≡ ⟨*request*.`host`, *requestUrl*.`protocol`⟩
↪ ∨ ⟨*request*.`host`, *requestUrl*.`protocol`⟩ ≡ *request*.`headers`[`Origin`]) **then**
18: **let** *origin* := *request*.`headers`[`Origin`]
19: **else**
20: **let** *origin* := ◇
21: **if** *response*.`status` ≡ 303 ∧ *request*.`method` ∉ {`GET`, `HEAD`} **then**
22: **let** *method′* := `GET`
23: **let** *body′* := ⟨⟩
24: **if** ∃ $\overline{w}$ ∈ Subwindows(*s′*) **such that** *s′*.$\overline{w}$.`nonce` ≡ π_2(*reference*) **then** → Do not redirect XHRs.
25: **let** *req* := ⟨`HTTPReq`, ν_6, *method′*, *url*.`host`, *url*.`path`, *url*.`parameters`, ⟨⟩, *body′*⟩
26: **let** *referrerPolicy* := *response*.`headers`[`ReferrerPolicy`]
27: **call** HTTP_SEND(*reference*, *req*, *url*, *origin*, *referrer*, *referrerPolicy*, *a*, *s′*)
28: **else**
29: **stop** ⟨⟩, *s′*
30: **switch** π_1(*reference*) **do**
31: **case** `REQ`
32: **let** $\overline{w}$ ← Subwindows(*s′*) **such that** *s′*.$\overline{w}$.`nonce` ≡ π_2(*reference*) **if possible;**
↪ **otherwise stop** → normal response
33: **if** *response*.`body` ≁ ⟨∗, ∗⟩ **then**
34: **stop** ⟨⟩, *s′*
35: **let** *script* := π_1(*response*.`body`)
36: **let** *scriptstate* := π_2(*response*.`body`)
37: **let** *d* := ⟨ν_7, *requestUrl*, *response*.`headers`, *referrer*, *script*, *scriptstate*, ⟨⟩, ⟨⟩, ⊤⟩
38: **if** *s′*.$\overline{w}$.`documents` ≡ ⟨⟩ **then**
39: **let** *s′*.$\overline{w}$.`documents` := ⟨*d*⟩
40: **else**
41: **let** $\overline{i}$ ← ℕ **such that** *s′*.$\overline{w}$.`documents`.$\overline{i}$.`active` ≡ ⊤
42: **let** *s′*.$\overline{w}$.`documents`.$\overline{i}$.`active` := ⊥
43: **remove** *s′*.$\overline{w}$.`documents`.($\overline{i}$ + 1) and all following documents
↪ from *s′*.$\overline{w}$.`documents`
44: **let** *s′*.$\overline{w}$.`documents` := *s′*.$\overline{w}$.`documents` +⟨⟩ *d*
45: **stop** ⟨⟩, *s′*
46: **case** `XHR`
47: **let** $\overline{w}$ ← Subwindows(*s′*), $\overline{d}$ **such that** *s′*.$\overline{d}$.`nonce` ≡ π_2(*reference*)
↪ ∧ *s′*.$\overline{d}$ = *s′*.$\overline{w}$.`activedocument` **if possible; otherwise stop**
→ process XHR response
48: **let** *headers* := *response*.`headers` − `Set-Cookie`
49: **let** *s′*.$\overline{d}$.`scriptinputs` := *s′*.$\overline{d}$.`scriptinputs` +⟨⟩
⟨`XMLHTTPREQUEST`, *headers*, *response*.`body`, π_3(*reference*)⟩
50: **stop** ⟨⟩, *s′*

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 20) 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 21) 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 57. *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:

Definition 64. The pair $((\langle a, f, m\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 22 (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 65 (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 23): The state of a document contains a term, say *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 *scriptinputs* and record which input it has already processed. For this purpose, the function CHOOSEINPUT$(s', scriptinputs)$ 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 *scriptinputs*. The index of this message is then saved in the scriptstate (which is returned to the script).

b) CHOOSEFIRSTINPUTPAT (Algorithm 24): Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term *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 *scriptinputs* that matches *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 PARENTWINDOW$(tree, docnonce)$ 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 *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 *docnonce*. If there is no such window (which is the case if the script runs in a document of a top-level window), PARENTWINDOW returns $\perp$.

d) PARENTDOCNONCE: The function PARENTDOCNONCE$(tree, docnonce)$ determines (similar to 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 *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 *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, PARENTDOCNONCE returns *docnonce*.

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

Algorithm 22 Web Browser Model: Main Algorithm.

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

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

55: **else if** $\exists \langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
$\hookrightarrow\ \pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
56: **let** $m' := \mathsf{dec_s}(m, key)$
57: **if** $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
58: **stop**
59: **remove** $\langle reference, request, url, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
60: **call** PROCESSRESPONSE(m', $reference$, $request$, url, a, f, s')
61: **else if** $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists \langle reference, request, url, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
$\hookrightarrow\ m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
62: **remove** $\langle reference, request, url, \bot, f\rangle$ **from** $s'.\texttt{pendingRequests}$
63: **call** PROCESSRESPONSE(m, $reference$, $request$, url, a, f, s')
64: **else if** $m \in$ DNSResponses **then** → Successful DNS response
65: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin$ IPs
$\hookrightarrow\ \vee\ m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].\texttt{request.host}$ **then**
66: **stop**
67: **let** $\langle reference, message, url\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
68: **if** $url.\texttt{protocol} \equiv \texttt{S}$ **then**
69: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
$\hookrightarrow\ +^{\langle\rangle} \langle reference, message, url, \nu_3, m.\texttt{result}\rangle$
70: **let** $message := \mathsf{enc_a}(\langle message, \nu_3\rangle, s'.\texttt{keyMapping}[message.\texttt{host}])$
71: **else**
72: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
$\hookrightarrow\ +^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
73: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
74: **stop** $\langle\langle m.\texttt{result}, a, message\rangle\rangle, s'$
75: **stop**

Algorithm 23 Function to retrieve an unhandled input message for a script.

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

f) AUXWINDOW: This function takes a term $tree$ and a document nonce $docnonce$ as input as above. From all window terms in $tree$ that have the window containing the document identified by $docnonce$ 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 $docnonce$.

g) AUXDOCNONCE: Similar to AUXWINDOW above, the function AUXDOCNONCE takes a term $tree$ and a document nonce $docnonce$ as input. From all window terms in $tree$ that have the window containing the document identified by $docnonce$ 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 $docnonce$.

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

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

j) GETORIGIN: To extract the origin of a document, the function GETORIGIN($tree$, $docnonce$) is used. This function searches for the document with the identifier $docnonce$ 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 $docnonce$ is found in the tree $tree$, $\diamond$ is returned.

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

Algorithm 24 Function to extract the first script input message matching a specific pattern.

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

Algorithm 25 Relation of a DNS server R^d.

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

J. DNS Servers

Definition 66. A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle\langle \texttt{domain}_1, a_1\rangle, \langle \texttt{domain}_2, a_2\rangle, \ldots\rangle \ .$$

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 $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of d above is defined by Algorithm 25.

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).

Definition 67. 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 68 (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 $\bot$ 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 26–30, and the main relation in Algorithm 31.

Algorithm 26 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

1: **function** HTTPS_SIMPLE_SEND($reference$, $message$, a, s')
2: **let** $s'.\texttt{pendingDNS}[\nu_{n0}] := \langle reference, message \rangle$
3: **stop** $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, \nu_{n0} \rangle\rangle\rangle$, s'

Algorithm 27 Generic HTTPS Server Model: Default HTTPS response handler.

1: **function** PROCESS_HTTPS_RESPONSE(m, $reference$, $request$, a, f, s')
2: **stop**

Algorithm 28 Generic HTTPS Server Model: Default trigger event handler.

1: **function** PROCESS_TRIGGER(a, s')
2: **stop**

Algorithm 29 Generic HTTPS Server Model: Default HTTPS request handler.

1: **function** PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2: **stop**

Algorithm 30 Generic HTTPS Server Model: Default handler for other messages.

1: **function** PROCESS_OTHER(m, a, f, s')
2: **stop**

Algorithm 31 Generic HTTPS Server Model: Main relation of a generic HTTPS server

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

let $s' := s$
if $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
  let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle, s'$
if $\exists\, m_{\text{dec}}, k, k', inDomain$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s.\texttt{tlskeys}$ **then**
  let $n, method, path, parameters, headers, body$ **such that**
  $\hookrightarrow\ \langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
  $\hookrightarrow$ **if possible; otherwise stop**
  call PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s')
else if $m \in$ DNSResponses **then** → Successful DNS response
  if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs}$
  $\hookrightarrow\ \vee\ m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].2.\texttt{host}$ **then**
    stop
  let $reference := s.\texttt{pendingDNS}[m.\texttt{nonce}].1$
  let $request := s.\texttt{pendingDNS}[m.\texttt{nonce}].2$
  let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, request, \nu_{n1}, m.\texttt{result}\rangle$
  let $message := \mathsf{enc_a}(\langle request, \nu_{n1}\rangle, s'.\texttt{keyMapping}[request.\texttt{host}])$
  let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
  stop $\langle\langle m.\texttt{result}, a, message\rangle\rangle, s'$
else if $\exists \langle reference, request, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
$\hookrightarrow$ **such that** $\pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
  let $m' := \mathsf{dec_s}(m, key)$
  if $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
    stop
  remove $\langle reference, request, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
  call PROCESS_HTTPS_RESPONSE(m', $reference$, $request$, a, f, s')
else if $m \equiv \texttt{TRIGGER}$ **then** → 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 69 (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

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

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 70. We say that a term t *is derivably contained in (a term)* t' *for (a set of DY processes)* P *(in a processing step* $s_i \to s_{i+1}$ *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 71. 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 72. We say that a DY process p *created* a message m in a processing step

$$(S^i, E^i, N^i) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S^{i+1}, E^{i+1}, N^{i+1})$$

of a run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ if all of the following hold true

- m is a subterm of one of the events in E_{out}
- m is and was not derivable by any other set of processes

$$m \notin d_\emptyset\Big(\bigcup_{\substack{p' \in \mathcal{W} \setminus \{p\} \\ 0 \leq j \leq i+1}} S^j(p')\Big)$$

We note a process p creating a message does not imply that p can derive that message.

Definition 73. 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 21).

Definition 74. 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 75. 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 76. We say that a *script initiated a request* r if a browser triggered the script (in Line 10 of Algorithm 20) and the first component of the *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 77. 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 31).

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

Lemma 20. 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

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

(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 $s_0.\mathtt{keyMapping}$ (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 preconditions.

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 22) that is not used anywhere else. Further, the key is stored in the browser's state in $pendingRequests$. The information from $pendingRequests$ 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 22). 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 22), a host header is always contained in HTTP requests by browsers. From Line 70 of Algorithm 22 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 $\mathtt{keyMapping}$ 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 20, 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 21. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\mathtt{windows}$ with the origin $\langle dom, \mathtt{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ 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 21) 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 21. 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 22. 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 $\mathtt{S}$. 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 22 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 $S_i(b).\mathtt{keyMapping}[dom] = S_0(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ 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 20 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 22. 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 $\langle dom, \mathtt{S} \rangle$ where $S_i(b).\mathtt{keyMapping}[dom] \equiv \mathsf{pub}(k)$ 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 21) 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 17), 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 20
- Line 72 of Algorithm 20
- Line 27 of Algorithm 21

■

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 21 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 21 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 20, but is applied to the generic HTTPS server (instead of the web browser).

Lemma 23. If in the processing step $s_i \to s_{i+1}$ 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 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 $s_j \to s_{j+1}$ where some process leaks k to $\mathcal{W} \setminus \{u, s\}$, there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, s\}$ 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 $s_j \to s_{j+1}$ 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 preconditions. The proof is the same as for Lemma 20:

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 $s_j \to s_{j+1}$ 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 31:

- The message emitted in Line 3 of Algorithm 26 has a different message structure
- As s is honest, it does not send the message of Line 6 of Algorithm 31
- There is no other place in the generic HTTPS server model where messages are emitted and due to precondition (V), the application-specific model does not emit HTTPS requests. ■

The value k, which is the placeholder ν_{n1} in Algorithm 31, is only stored in the $pendingRequests$ subterm of the state of s, i.e., in $S^{i+1}(s)$.`pendingRequests`. Other than that, s only accesses this value in Line 19 of Algorithm 31, 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 precondition (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 31 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 `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 precondition (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 31, 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 26, where a DNS request is sent, and in Line 18 of Algorithm 31, 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 31 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'.