Formal Security Analysis of the OpenID FAPI 2.0: Accompanying a Standardization Process

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

Abstract

In recent years, the number of third-party services that can access highly-sensitive data has increased steadily, e.g., in the financial sector, in eGovernment applications, or in high-assurance identity services. Protocols that enable this access must provide strong security guarantees.

A prominent and widely employed protocol for this purpose is the OpenID Foundation's FAPI protocol. The FAPI protocol is already in widespread use, e.g., as part of the UK's Open Banking standards and Brazil's Open Banking Initiative as well as outside of the financial sector, for instance, as part of the Australian government's Consumer Data Rights standards.

Based on lessons learned from FAPI 1.0, the OpenID Foundation has developed a completely new protocol, called FAPI 2.0. The specifications of FAPI 2.0 include a concrete set of security goals and attacker models under which the protocol aims to be secure.

Following an invitation from the OpenID Foundation's FAPI Working Group (FAPI WG), we have accompanied the standardization process of the FAPI 2.0 protocol by an in-depth formal security analysis. In this paper, we report on our analysis and findings.

Our analysis incorporates the first formal model of the FAPI 2.0 protocol and is based on a detailed model of the web infrastructure, the Web Infrastructure Model, originally proposed by Fett, Küsters, and Schmitz. Our analysis has uncovered several types of attacks on the protocol, violating the aforementioned security goals set by the FAPI WG. We subsequently have worked with the FAPI WG to fix the protocol, resulting in several changes to the specifications. After adapting our model to the changed specifications, we have proved the security properties to hold under the strong attacker model defined by the FAPI WG.

This technical report embodies the full version of our CSF 2024 paper with the same title.

I. Introduction

Web-based authentication and authorization is ubiquitous. Many websites and applications can be used by logging in with a so-called Identity Provider, for instance, "Login with Google" or "Login with Facebook", generally dubbed "social login", or more generally, *Single Sign-On (SSO)*. It is also possible to authorize applications, including websites, but also IoT devices, such as routers and smart TVs, to access resources managed by the Identity Provider. Such resources include email addresses [24, 74], documents/calendars/pictures/movies stored in the cloud [59, 61], YouTube accounts [81], or development repositories [74]. A widely used protocol family for these authorization and authentication use cases are the OAuth 2.0 and OpenID Connect protocols [79, 82].

While plain OAuth 2.0 and OpenID Connect are suitable for typical low-risk use cases (e.g., social login), many use cases have emerged in high-risk settings for both authorization and authentication scenarios: Third party services can be authorized to, e.g., get access to bank transaction histories for monitoring and feedback [4, 10], trigger financial transactions [62, 63, 80], access cars [12], perform health-related actions like managing electronic prescriptions [23], or access medical records [19, 76]. In such high-risk use cases, attacks that enable malicious actors to access resources or impersonate end-users not only have more severe consequences than in classical low-risk settings, but such use cases also require more robust protocols and overall stronger security guarantees. For example, when using SSO to manage access to health records, it is important to not only recognize the same user again at a later point, which is typical for social network SSO, but to provide the full legal identity of the user. Likewise, protocols for high-risk use cases should be robust, i.e., provide security even if some messages or relevant values leak to an attacker, e.g., through leaked server log files [1].

To provide a higher level of security for such use cases, the OpenID Foundation developed FAPI 1.0, which is based on OAuth 2.0 and OpenID Connect, but uses many additional mechanisms to increase their security, e.g., to guarantee authorization and authentication even if the attacker can misconfigure certain endpoints or certain TLS-protected messages leak to the attacker, e.g., via log files. Meeting many ecosystems' needs, FAPI 1.0 is now in widespread use, e.g., as part of the UK's Open Banking standards [62], in the Australian Government's (mandatory to support) *Consumer Data Standards* [16] which govern customer interaction with banks, energy, and telecommunications companies, with more industry sectors to follow. Further FAPI 1.0 uses include Brazil's Open Finance and Open Insurance programs [63, 64], companies like yes.com [80] with an ecosystem of more than 1,000 actively participating banks, acting as trusted identity providers that can be used by sites and apps to login users, identify natural persons, obtain account information, and initiate payments, as well as US-based Financial Data Exchange FDX with more than 42 million users [27], and New Zealand's core payment clearing house payments.nz [40].

The high security goals FAPI 1.0 aims to achieve have been formally analyzed before [38], uncovering several attacks and proposing fixes for those. Based on the experiences with FAPI 1.0, including interoperability and implementation aspects, the OpenID Foundation is currently standardizing a successor named FAPI 2.0, which comprises a completely new protocol (see also Section IV-C).

FAPI 2.0 is a framework of specifications, with the core protocol specified in the *FAPI 2.0 Security Profile* [32]. Another important specification for our purposes in this framework is the *FAPI 2.0 Attacker Model* [29], which captures the security goals that the protocol aims to fulfill, along with assumptions on the attacker capabilities, resulting in a strong attacker, far exceeding standard attacker models for protocol analysis (see Section II-C). In the following, we will often refer to the FAPI 2.0 Security Profile and FAPI 2.0 Attacker Model as just FAPI 2.0.[1]

Despite being a very recent standard, FAPI 2.0 is expected to be adopted soon in many important ecosystems, with several of the aforementioned FAPI 1.0 users already having committed to switch to FAPI 2.0.

Given the importance of FAPI 2.0 and its current and future use in high-risk environments, the FAPI WG has asked us to accompany the standardization process with a formal analysis, providing feedback early on and throughout the development of FAPI 2.0. We hence have performed a detailed formal security analysis of the FAPI 2.0 Security Profile with the security goals specified in the FAPI 2.0 Attacker Model. Our analysis is based on the Web Infrastructure Model (WIM) [33], a symbolic Dolev-Yao style model of the Web infrastructure. The WIM is the most comprehensive and detailed model of the Web infrastructure to date. Other models of the Web and its infrastructure, such as those by Pai et al. [65], Kumar [49, 50], or Bansal et al. [6] are significantly more abstract and limited by the tools upon which they are based.

For our work, we used the WIM as-is in its most recent published version [51], except for an additional HTTP header for DPoP proofs (see below for DPoP and Appendix D-I).

For the analysis of FAPI 2.0, we have created a formal model of the FAPI 2.0 Security Profile and formalized the security properties stated in the FAPI 2.0 Attacker Model and incorporated the assumptions on the attacker laid out therein (based on the state of the specifications at that time). We coordinated these steps with the FAPI WG to ensure a faithful modeling. The process of proving the properties within the model has revealed several attacks that break the goals of the protocol. We have proposed fixes and improvements to the specifications, which the FAPI WG appreciated and amalgamated into the official specifications, resulting in substantial changes thereof.

We adapted our formal model and formal security properties according to the changes discussed with the FAPI WG and were finally able to prove that the desired properties hold true within the model. Hence, our analysis reflects the latest official version of the FAPI 2.0 specifications [29, 32]. We note that while our analysis uncovered new attacks, we also found known attack patterns which the FAPI WG is familiar with and tried to avoid. This highlights the importance of a systematic formal analysis, which makes it possible to detect subtle flaws even in very complex protocols, where it is easy to overlook such flaws.

Contributions. In summary, our contributions are:

- We provide the first formal model of FAPI 2.0, along with a formalization of the security goals set forth in the FAPI 2.0 Attacker Model.
- Our analysis has uncovered several attacks, i.e., violations of the security goals under the attacker model defined by the FAPI WG.
- We propose fixes and improvements and worked with the FAPI WG to incorporate them, resulting in significantly modified and improved FAPI 2.0 specifications.
- We adapted our formal model to reflect the improved specifications and were then able to prove the formalized security goals.
- We have accompanied the development of the specifications from an early stage on and were able to support the standardization process with security recommendations before widespread deployment of the protocol.

Structure of This Paper. We first give a detailed description of the FAPI 2.0 protocol in Section II and then present the attacks that we have discovered in the process of our formal analysis in Section III. We describe our formal model and formal security theorem along with a proof sketch in Section IV. The full formal model and proofs are given in the appendix. Related work is discussed in Section V. We conclude in Section VI.

II. FAPI 2.0 Protocol and Security Goals

In the following, we describe the FAPI 2.0 protocol and the accompanying FAPI 2.0 Attacker Model as of the beginning of our work. Recall that the FAPI 2.0 Attacker Model is part of the FAPI 2.0 specification framework. After consultation with the FAPI WG, we used FAPI 2.0 specifications as of June 1st, 2022 as a basis for our analysis ([30, 31], the FAPI 2.0 Security Profile used to be called *Baseline Profile*). We discuss changes to the specifications made due to our findings since then in Section III.

[1]While in the original FAPI 1.0 protocol, "FAPI" stands for "Financial-grade API", the scope and expected uses of FAPI 2.0 reach far beyond the financial sector, thus, FAPI 2.0 is not an acronym anymore.

A. Overview of FAPI 2.0

In a nutshell, FAPI 2.0 allows a user (also called resource owner) to grant a *client* application access to their data stored at a *resource server (RS)*, by means of an *authorization server (AS)* which is responsible to manage access to the user's data. In addition, the AS may provide the client with information on the user's identity at the AS. For example, FAPI 2.0 may be used to grant an account aggregation service (client) read access to a user's account balance at various banks (RSs), with services of these banks (ASs) managing such access (such services are in use today, e.g., [4, 10, 62, 63]).

On a high level, a FAPI 2.0 protocol run, also called *flow* or *grant*, advances as follows: A user visits a website or uses an application of the client c, which wants to access data of the user stored at the RS. Since the user's data at the RS is managed by an AS AS, c contacts AS with some initial information, e.g., what kind of data c want to access. AS replies with an internal reference to the current flow, which c then forwards to the user's browser while also instructing the browser to visit a website of AS to proceed. Once the user, or more precisely, their browser, visits that AS website, the user is asked to authenticate, e.g., with username and password, and to authorize the client's request. If the user consents, AS instructs the user's browser to return to the client website or application, passing on a value called the *authorization code*. Once the client receives that authorization code, it can contact AS and exchange the authorization code for so-called tokens. There are two types of tokens in FAPI 2.0: *ID Tokens* and *Access Tokens*. An id token contains information to identify the user, e.g., an email address or username with which the user is registered at the AS. This id data can be used by the client to authenticate users in the context of the client application. An access token, on the other hand, can be used by the client to request users' resources from an RS, e.g., account balances. Upon receiving such a request, an RS verifies the access token's validity. Depending on the access token format, this may include checking a signature on the access token or using so-called *token introspection*, which means that the RS queries the AS for validity information on a given access token.

B. The FAPI 2.0 Protocol in Detail

In the following, we describe a FAPI 2.0 protocol flow in detail (depicted in Figure 1). The flow is initiated by a user visiting the website or using an application of a client c, typically expressing the wish to authorize c using a certain AS AS, e.g., by clicking a "Login with AS" button (Step 1).

FAPI 2.0 assumes that c received the so-called *issuer identifier* $iss_{\texttt{AS}}$ of AS (e.g., via configuration). That issuer identifier is used in FAPI 2.0 and other protocols to uniquely identify AS [75]. However, to complete a FAPI 2.0 flow, c needs additional knowledge on AS, e.g., endpoint URLs. If c does not yet know all necessary values (e.g., via configuration), it can proceed by fetching so-called *Authorization Server Metadata* [48, 71] from AS (Step 2). However, this step is optional. Like all other communication in FAPI 2.0, this exchange is done via HTTPS, i.e., is protected by TLS. The metadata returned by AS includes URIs of the relevant endpoints, supported cryptographic algorithms, and similar information, along with the issuer identifier of AS (Step 3). Once the client acquired the metadata, it verifies the aforementioned issuer identifier to prevent mix-ups, e.g., due to injection attacks.

Once the required values are available, c assembles a *Pushed Authorization Request* (PAR) [57] and sends it to AS (Step 5). This PAR request contains everything needed by AS to provide the user with sufficient information in Step 10 such that the user can make an informed decision on whether to grant c access to their data. This information includes: 1) a *client id* cid, uniquely identifying c at AS. 2) A *scope* value, describing what data c wants to access, e.g., "read transactions", and whether c requests an id token to be issued. 3) A *redirect uri* $rediruri$, which is used by AS in Step 11 to redirect the user's browser back to c. 4) A *code challenge*, i.e., a hash $h(cv)$ of a client chosen nonce cv, which is used in Step 14 to verify that the client requesting a token is the same client that sent the PAR request (even if the PAR request leaks). This mechanism is called *Proof Key for Code Exchange* (PKCE) [72]. 5) Client authentication information (see below for a description).

Upon receiving the PAR request, AS verifies the client authentication, the presence of the parameters explained above, and checks whether the requested scope can be granted to the client (under the policies of AS). If all these checks pass, AS creates a random *request uri* $requri$ and stores the requested scope, cid, $cc := h(cv)$, $rediruri$, and $requri$ (Step 6); $requri$ will be used as a reference to the PAR data in Step 9 and is therefore sent to c in the PAR response (Step 7). Client c then redirects the user's browser to AS, adding $requri$ and cid as request parameters (Step 8). Following that redirect, the user's browser visits AS and in doing so, forwards $requri$ and cid, hence providing information on the user's context (i.e., the current flow) to AS (Step 9). The user now authenticates at AS and reviews the access requested by c (Step 10), the exact details of this step are up to the AS and out of scope of FAPI 2.0. If the user consents, AS generates a random *authorization code* ac and stores it with the PAR data from Step 5. AS then redirects the user's browser back to the $rediruri$ of c (stored in Step 6), and includes ac as well as an iss value [75] (i.e., the issuer identifier $iss_{\texttt{AS}}$) as parameters (Steps 11 and 12).

Once c has received the browser's (redirected) request, it validates the iss value by comparing it to the issuer identifier of the AS to which c sent the PAR request in Step 5 to prevent mix-up attacks [36, 55, 58, 75]. If this check passes, c sends a *token request* to AS (Step 14). This token request contains the authorization code ac from Step 12, client id cid, a *code verifier* cv, i.e., the nonce from Step 5, and client authentication similar to Step 5. Furthermore, c must include information for access token sender constraining, which we describe below.

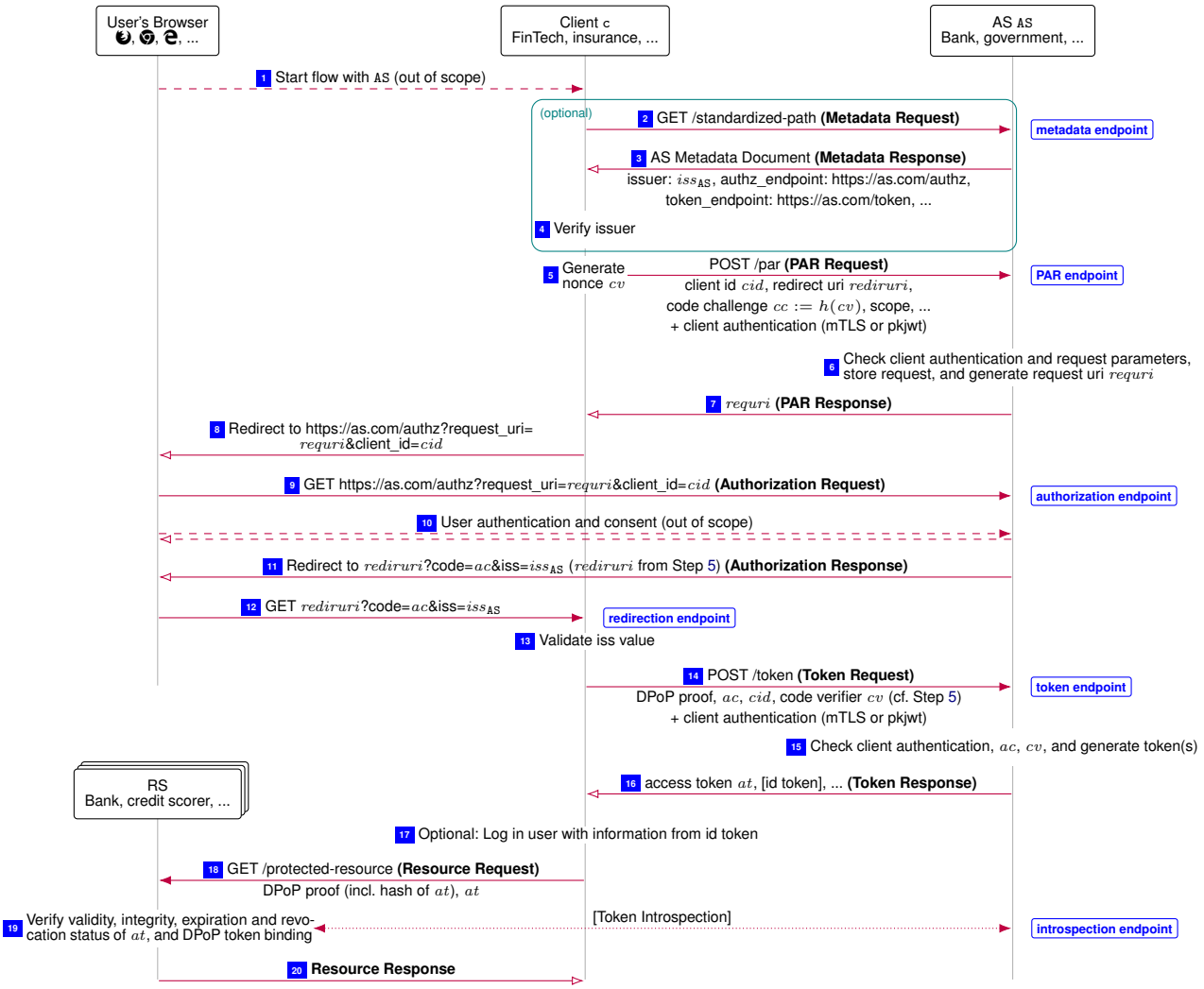

Figure 1. FAPI 2.0 Security Profile protocol flow (with DPoP sender constraining)

When AS receives that token request, it verifies the client authentication, presence of a sender constraining method, and validity of the authorization code and code verifier (Step 15). The latter is verified by checking whether $h(cv) = cc$, with cc being the code challenge stored in Step 6 and cv being the code verifier from the token request. The code ac is then invalidated and AS generates an access token at (and id token if requested) and sends them back to c in Step 16.

Given an id token, c may now log in the user with whatever identity the user has at AS, e.g., a user name (Step 17). This allows clients to offer SSO to their users.

Using the access token at, c can request user's resources at an RS as follows: in the resource request (Step 18), c must include at as well as corresponding information for access token sender constraining (see below). The RS then has to verify at's validity, integrity, expiration, and revocation status, as well as the sender constraining information (Step 19). Except for the sender constraining, FAPI 2.0 does not specify how RSs should perform those (nonetheless mandatory) checks. Currently, there are two widely-adopted methods to do so [60]: token introspection [69], and structured access tokens, which contain the necessary information and are typically signed by the AS [8, 46]. With token introspection, the RS sends the access token to the introspection endpoint of the AS which issued the token, to which the AS answers with information on the validity of the token and on the public key to which the access token is bound.

Client Authentication. FAPI 2.0 requires ASs to authenticate clients at the PAR and token endpoints (Steps 5 and 14) using *Mutual-TLS (mTLS)* or `private_key_jwt`. In both cases, clients need to be registered with the AS beforehand. With mTLS [11], the client presents a TLS certificate containing the client's identity, e.g., its client id, during TLS connection establishment. With `private_key_jwt` [70], the client adds a signed *JSON Web Token (JWT)* [44, 46, 47] to its messages. This JWT contains the client's id at the AS, the issuer identifier of the AS, and a nonce, and is signed with a private key of the client.

Access Token Sender Constraining. When issuing an access token (Steps 14–16), a FAPI 2.0 AS is required to bind the

token to a key of the client who requested it. Likewise, the RS must verify this binding when it receives a resource request (Step 19). FAPI 2.0 defines two methods to establish and verify such a binding: *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)* [39], which is shown in Figure 1, and mTLS [11]. In both cases, the access token is bound to a client key pair, e.g., by including a hash of the public key in the token, and the client has to include a proof of possession of the private key when using the access token.

With DPoP, the token request (Step 14) must include a *DPoP proof*: a signed JWT $dpopJWT$, containing the URL to which it is sent, a nonce, and a public verification key $\mathtt{pub}(k)$ (of the client's choice). $dpopJWT$ is signed using the corresponding private key k. The AS then binds the access token to $\mathtt{pub}(k)$. When requesting resources (Step 18), the client has to include another DPoP proof—signed with k—which must contain a hash of the access token in addition to the aforementioned items.

With mTLS, the AS binds the access token to the public key included in the client's TLS certificate, which the client presents during connection establishment in Step 14. When using the access token (Step 18), the client presents the same certificate during the TLS connection establishment (which includes a proof of possession of the corresponding private key).

We emphasize that client authentication and access token sender constraining are independent of each other, including the key material. E.g., a client which uses mTLS to authenticate may use DPoP for sender constraining, and a client can authenticate with `private_key_jwt` and at the same time use mTLS for sender constraining. I.e., there are four possible combinations.

C. Security Goals and Attacker Model

Along with the actual protocol specification, the FAPI WG developed the FAPI 2.0 Attacker Model [30] which outlines security goals and assumptions on attackers under which these goals are expected to hold. As before, we describe the state as of June 1st, 2022 here and discuss changes made since then in Section III. The formalized security properties and modeling of attacker assumptions are presented in Section IV.

Authorization Goal. The authorization goal states that no attacker should be able to access resources belonging to an honest user. In addition, the FAPI 2.0 Attacker Model states that this goal is "fulfilled if no attacker can successfully obtain and use an access token" issued for an honest user.

Authentication Goal. The authentication goal is fulfilled when no attacker is able to log in at a client under the identity of an honest user.

Session Integrity for Authorization Goal. Session integrity goals aim to prevent attackers from tricking users into using attacker's resources or identities. Hence, the session integrity for authorization goal ensures users cannot be forced to use resources of the attacker.

Session Integrity for Authentication Goal. Similar to the session integrity for authorization goal, the session integrity for authentication goal is fulfilled if no attacker can force an honest user to be logged in under an identity of the attacker.

Attacker Assumptions. In the following, we summarize the aforementioned attacker assumptions laid out in the FAPI 2.0 Attacker Model. We stress that these (strong) assumptions are part of and justified by the specification.

$\mathcal{A}1$. The attacker controls the network, i.e., can intercept, block, and tamper with all messages sent over the network. In particular, the attacker can also reroute, reorder, and create (from its knowledge) new messages. However, the attacker cannot break cryptography unless it learned the respective keys. Nevertheless, the attacker can pose as any party (and any network participant) in the protocol. In addition, the attacker can also send links to (honest) users which are then visited by these users. See [30, Sec. A1, A1a, A2].

$\mathcal{A}2$. The attacker can read authorization requests in plain (cf. Step 9 in Figure 1). See [30, Sec. A3a].

$\mathcal{A}3$. The attacker can read authorization responses in plain (Step 12). See [30, Sec. A3b].

$\mathcal{A}4$. The attacker can trick the client into using an attacker-controlled token endpoint URL, i.e., one for which the attacker can obtain a valid TLS certificate (other endpoints, e.g., PAR, are not affected). Hence, the attacker can read token requests (Step 14) in plain and construct arbitrary token responses from its knowledge (Step 16). However, this assumption only applies to clients which do not use the AS metadata mechanism. See [30, Sec. A5].

$\mathcal{A}5$. Resource requests (Step 18) leak to the attacker in plain. See [30, Sec. A7].

$\mathcal{A}6$. Resource responses (Step 20) leak to the attacker in plain. See [30, Sec. A7].

$\mathcal{A}7$. The attacker can modify resource responses (Step 20). I.e., the attacker can replace an honest RS' resource response with its own message without the client noticing, even though that response is protected by TLS. Note that this does *not* give the attacker the ability to replace arbitrary messages in TLS connections, but is limited to resource responses. See [30, Sec. A8].

III. Attacks

We formally modeled the FAPI 2.0 specifications (as of June 1st, 2022) and then formalized and tried to prove the security goals according to the FAPI 2.0 Attacker Model [30] under the attacker assumptions outlined in the same document. In the course of this analysis, we have uncovered a number of attacks, i.e., violations of the security goals laid out in the FAPI 2.0 Attacker Model. We have discussed these findings with the FAPI WG and worked with them to resolve the issues, resulting in a number of changes to the specifications which we explain here. The formal model presented in Section IV, for which we prove security, incorporates

these changes. While, to the best of our knowledge, Attacker Token Injection, the Client Impersonation attacks, and DPoP Proof Replay are completely new attacks, interestingly, for the other attacks (Browser Swapping, Cuckoo's Token, Authorization Request Leak), similar attack patterns have been reported for related protocols [38, 54]. This emphasizes the importance of systematic, formal analysis, as even seasoned experts overlook known attack patterns for complex protocols.

Due to space constraints, we present additional attacks and their fixes as well as inconsistencies in the specifications discovered during our analysis in Appendix B.

A. Attacker Token Injection

This two-phased attack violates both session integrity goals and requires attacker assumptions $\mathcal{A}4$ (token endpoint misconfiguration) and $\mathcal{A}5$ (resource requests leak, see Section II-C). In the first phase, the attacker, posing as a user, completes a flow with an honest client $\mathsf{c}_{\mathsf{hon}}$ and honest AS $\mathtt{AS}_{\mathsf{hon}}$. During this flow, the attacker uses $\mathcal{A}5$ to obtain at_{att}, i.e., an access token bound to keys of $\mathsf{c}_{\mathsf{hon}}$, issued by $\mathtt{AS}_{\mathsf{hon}}$ for resources of the attacker. For the second phase, the attacker uses $\mathcal{A}4$, such that $\mathsf{c}_{\mathsf{hon}}$ uses an attacker-controlled token endpoint (instead of $\mathtt{AS}_{\mathsf{hon}}$'s token endpoint). Hence, when an honest user u starts a flow with $\mathsf{c}_{\mathsf{hon}}$ and $\mathtt{AS}_{\mathsf{hon}}$, the attacker receives $\mathsf{c}_{\mathsf{hon}}$'s token request, to which the attacker answers with at_{att} and an id token constructed by the attacker for an attacker identity. Client $\mathsf{c}_{\mathsf{hon}}$ then logs u in under the attacker's identity and uses attacker resources in a session with u.

At its core, this attack is possible because the attacker can trick an honest client into using an attacker-controlled token endpoint (due to $\mathcal{A}4$). We describe a fix, our communication with the FAPI WG, and resulting changes to the specifications at the end of the next subsection.

B. Client Impersonation Attacks

This attack violates the authorization goal and requires attacker assumption $\mathcal{A}4$ (token endpoint misconfiguration, see Section II-C). The attack targets an honest client $\mathsf{c}_{\mathsf{hon}}$, which authenticates itself to an honest AS $\mathtt{AS}_{\mathsf{hon}}$ with the `private_key_jwt` method (see Section II-B). Using $\mathcal{A}4$, the attacker modifies $\mathsf{c}_{\mathsf{hon}}$'s configuration such that $\mathsf{c}_{\mathsf{hon}}$ uses an attacker-controlled token endpoint (instead of $\mathtt{AS}_{\mathsf{hon}}$'s token endpoint).

In a nutshell, the attacker first obtains two valid client authentication JWTs $pkjwt_1$ and $pkjwt_2$ for $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$. Afterwards, the attacker uses those JWTs to impersonate $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$ and obtains an access token issued by $\mathtt{AS}_{\mathsf{hon}}$ for resources of an honest user u, with the token being bound to a key of the attacker, i.e., the attacker can use this token at an RS to access u's resources.

To obtain said JWTs, the attacker starts a flow with $\mathsf{c}_{\mathsf{hon}}$, selects $\mathtt{AS}_{\mathsf{hon}}$, and authenticates at $\mathtt{AS}_{\mathsf{hon}}$ (with an attacker id). However, once the attacker receives $\mathsf{c}_{\mathsf{hon}}$'s token request (due to the misconfigured token endpoint), no further actions are taken. The token request sent by $\mathsf{c}_{\mathsf{hon}}$ contains $pkjwt_1$ to authenticate $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$, which has now leaked to the attacker and has never been sent to $\mathtt{AS}_{\mathsf{hon}}$, i.e., $pkjwt_1$ is still valid. The attacker can repeat this to obtain $pkjwt_2$.

With such valid client authentication JWTs for $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$, the attacker can proceed as depicted in Figure 2: An honest user u starts a flow with $\mathsf{c}_{\mathsf{hon}}$ and expresses their wish to use an AS $\mathtt{AS}_{\mathsf{att}}$, identified by issuer identifier iss_{att}, which happens to be controlled by the attacker (Step 2). Hence, $\mathsf{c}_{\mathsf{hon}}$ sends a PAR request to $\mathtt{AS}_{\mathsf{att}}$, containing its client id cid at $\mathtt{AS}_{\mathsf{att}}$, a redirect uri, a code challenge $h(cv_{\mathsf{hon}})$ for a code verifier cv_{hon} chosen by $\mathsf{c}_{\mathsf{hon}}$, etc. as described in Section II-B (Step 3). Instead of replying to this request immediately, the attacker now poses as $\mathsf{c}_{\mathsf{hon}}$ towards $\mathtt{AS}_{\mathsf{hon}}$: the attacker sends its own PAR request to $\mathtt{AS}_{\mathsf{hon}}$, assembled from an attacker-chosen code challenge $h(cv_{\mathsf{att}})$, a redirect uri $rediruri_{\mathsf{att}}$ pointing to a URL of $\mathtt{AS}_{\mathsf{att}}$, the client id cid' of $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$, and $pkjwt_1$ (Step 4). Upon receiving the attacker's PAR request, $\mathtt{AS}_{\mathsf{hon}}$ validates the request and replies with $requri$ (Step 5), which the attacker forwards to $\mathsf{c}_{\mathsf{hon}}$ (now again in the role of $\mathtt{AS}_{\mathsf{att}}$ towards $\mathsf{c}_{\mathsf{hon}}$) in Step 6 in response to $\mathsf{c}_{\mathsf{hon}}$'s original PAR request from Step 3.

Client $\mathsf{c}_{\mathsf{hon}}$ now instructs u to visit $\mathtt{AS}_{\mathsf{att}}$ for authentication (Step 7). However, instead of the usual login page, $\mathtt{AS}_{\mathsf{att}}$ responds with a page luring the user into clicking a link, e.g., by explaining that it is now cooperating with $\mathtt{AS}_{\mathsf{hon}}$ (Step 9). This link points to $\mathtt{AS}_{\mathsf{hon}}$'s authorization endpoint and contains the parameters $requri$ and cid', i.e., the client id of $\mathsf{c}_{\mathsf{hon}}$ at $\mathtt{AS}_{\mathsf{hon}}$; instead of a link, $\mathtt{AS}_{\mathsf{att}}$ could also just redirect u to $\mathtt{AS}_{\mathsf{hon}}$ directly. Hence, u ends up authenticating at $\mathtt{AS}_{\mathsf{hon}}$ and authorizes $\mathsf{c}_{\mathsf{hon}}$ – recall that u *expects* to authorize $\mathsf{c}_{\mathsf{hon}}$ here (Step 11). Following u's consent, $\mathtt{AS}_{\mathsf{hon}}$ redirects u with authorization code ac to $rediruri_{\mathsf{att}}$ received in Step 4, i.e., to an attacker-controlled location (Step 12), which u follows (Step 13).

Having received ac, the attacker can now construct a valid token request (using $pkjwt_2$) as shown in Step 14, and subsequently receives an access token at for u's resources. Note that from $\mathtt{AS}_{\mathsf{hon}}$'s point of view, u authorized the attacker, posing as $\mathsf{c}_{\mathsf{hon}}$ towards $\mathtt{AS}_{\mathsf{hon}}$, to receive at. In addition, recall that the DPoP key to which $\mathtt{AS}_{\mathsf{hon}}$ binds at is chosen by the sender of the token request (Step 14), i.e., the attacker. Hence, at is bound to a key of the attacker, and can be used at an RS to access u's resources, thus breaking the authorization goal.

We stress that u authenticates at an honest, trusted AS and authorizes not only an honest and trusted client, but also exactly the client u expected to authorize. There are some variants of this attack with slightly different preconditions, but similar outcome, which we describe in Appendix B-B.

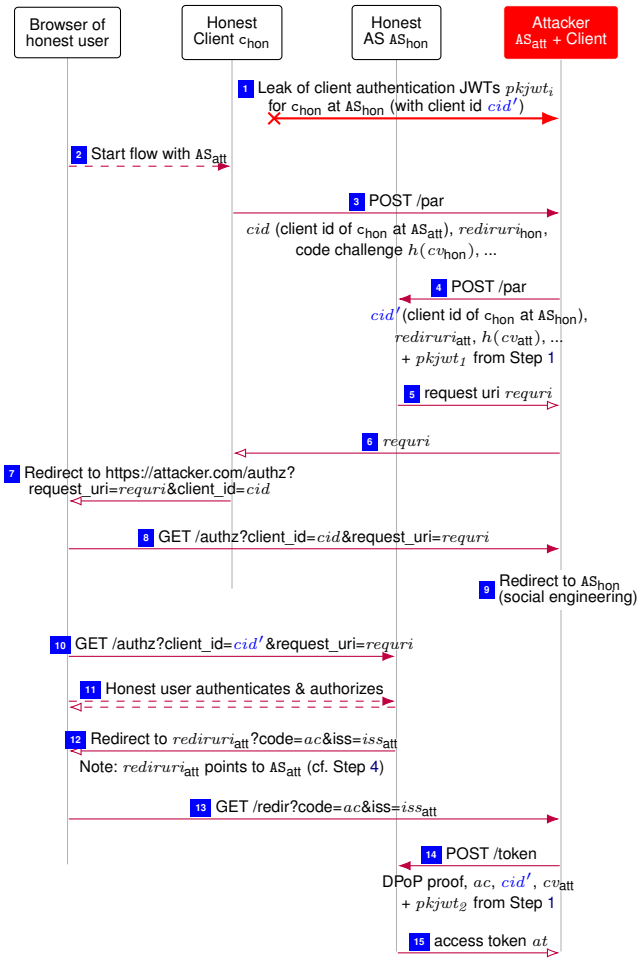

Figure 2. Client impersonation attack

These attacks emerged when we tried to prove that a client authentication JWT for authentication of an honest client at an honest AS AS_{hon} (i.e., the JWT contains the issuer identifier of AS_{hon}) cannot leak to an attacker (Lemma 9).

Fix. Since the possible misconfiguration of an honest client's token endpoint is the root cause for both the attacker token injection and the client impersonation attacks and FAPI 2.0 ASs are already required to serve a metadata document, our proposed fix of mandating clients to request and use this metadata was adopted by the FAPI WG [20, 25, 42]. Recall $\mathcal{A}4$: such token endpoint misconfiguration is only considered for clients which do not use the AS metadata mechanism.

C. DPoP Proof Replay

With attacker assumption $\mathcal{A}5$ (resource requests leak) from Section II-C, the attacker can read resource requests in plain and hence can try to replay them at the RS (cf. Step 18 in Figure 1), thus violating the authorization goal (see Section II-C) if the RS accepts the replayed request.

When DPoP sender constraining is used, the attacker can indeed replay the client's DPoP proof (using the attacker's TLS keys for the underlying connection): neither FAPI 2.0, nor DPoP [39] itself, nor the specifications on which DPoP is built [45, 46, 70] mandate for DPoP proofs to be strictly one-time use, hence, the RS does not reject the replayed proof.

Our initial attempts to prove the authorization property (see Definition 2) revealed a second variant, which also violates the authorization goal: with $\mathcal{A}1$ (network attacker), the attacker can additionally block the honest client's request, i.e., from the point of view of the RS, the attacker's resource request is not even a replay, and hence, replay protection does not completely prevent this attack.

Note that either variant of this attack is not possible with mTLS sender constraining since the attacker cannot even establish an mTLS connection with the client's public mTLS key (to which the access token is bound when mTLS sender constraining is used).

Fix. In our discussions with the FAPI WG, it became clear that the FAPI WG formulated attacker assumption $\mathcal{A}5$ with leaks of RS server log files in mind. Hence, the FAPI 2.0 Attacker Model was changed to clarify that resource requests leak *after* processing by the RS [13]. However, this only resolves the problem resulting from the attacker blocking the honest client's request, but does

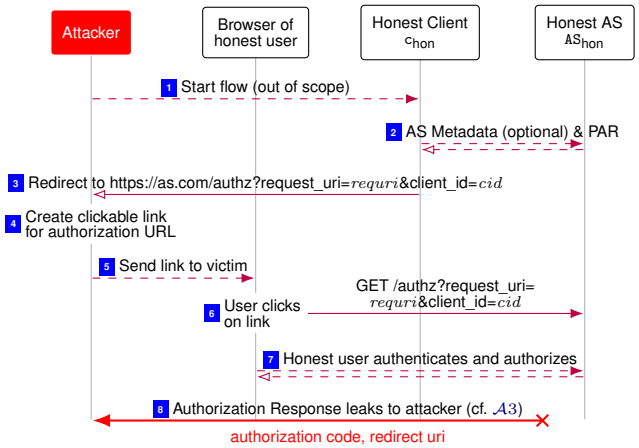

Figure 3. Browser swapping attack

not prevent the replay attack. Hence, we proposed to fix this attack by mandating the use of *resource server-provided nonces* with strict one-time use enforcement by the RS [39, Sec. 9]. The server-provided nonce mechanism is an optional part of DPoP in which—in a challenge-response manner—the client first requests a nonce from the RS which the client then has to include in its DPoP proof. We validated effectiveness of this fix in our formal model.

The FAPI WG acknowledged the attack [22], and added a description of the attack, as well as several options to fix it, to the specification.

D. Browser Swapping Attack

In this attack, the attacker violates the authorization and authentication goals by combining attacker assumptions $\mathcal{A}1$ (network attacker) and $\mathcal{A}3$ (authorization responses leak, see Section II-C).

On a high level, the attacker poses as a user and starts a flow with an honest client c_{hon} and honest AS AS_{hon}, but tricks an honest user u into logging in at AS_{hon} and to authorize access for c_{hon}. After u authorized c_{hon}, the attacker continues its session with c_{hon} (as if the attacker authenticated and authorized at AS_{hon}). Hence, the attacker gets logged in at c_{hon} under the identity of u, and c_{hon} provides access to u's resources to the attacker.

The detailed attack flow is depicted in Figure 3: in Step 1, the attacker initiates a flow at c_{hon} with AS_{hon}. Then, c_{hon} and AS_{hon} exchange the PAR as described in Section II-B (Step 2). Following this, c_{hon} instructs its user, i.e., the attacker, to visit the authorization endpoint of AS_{hon} with the request uri $requri$ from Step 2 and c_{hon}'s client id cid. However, instead of following this redirect, the attacker creates a clickable link, pointing to AS_{hon}'s authorization endpoint, with $requri$ and cid as parameters (Step 4). The attacker then sends this link to u in Step 5, e.g., in an email or as part of an attacker website.

Once u follows that link (e.g., by means of social engineering, see $\mathcal{A}1$) in Step 6, u will be asked to authenticate (if not already logged in at AS_{hon}) and to authorize c_{hon} to access u's resources (Step 7). We emphasize that, similar to AS_{hon}, u might trust the (honest) client c_{hon}. Once u consents, the attacker blocks all further communication for u (see $\mathcal{A}1$).

Now recall $\mathcal{A}3$: through a leaking authorization response, the attacker can obtain the authorization code ac (Step 8). With ac, the attacker visits c_{hon}'s redirect uri, so c_{hon} can subsequently exchange ac for an access (and id) token at AS_{hon}. From c_{hon}'s point of view, these tokens are associated with c_{hon}'s session with the attacker. However, due to Step 7, these tokens are issued for the (honest user) u's resources (and identity).

At the heart of this attack is the lack of a strong connection between the sessions user–client and user–AS. We discovered it when trying to prove the authorization property (see Definition 2): while proving that in a flow between an honest client, honest AS, and honest user, the client does not leak the user's resources, we have to prove, among others, that the authorization code associated with the flow cannot be sent to the client's redirection endpoint by the attacker.

Fix. The FAPI WG acknowledged this attack and after several discussions, there was consensus that this attack cannot be fixed with currently deployed methods [9, 66]. This decision is documented and explained along with the attack in the specifications [29, Sec. 6.5.7]. Consistent with this decision, the FAPI WG also removed attacker assumption $\mathcal{A}3$, i.e., FAPI 2.0 no longer claims to fulfill its security goals when authorization responses leak.

E. Cuckoo's Token Attack

In this attack, the attacker leverages attacker assumption $\mathcal{A}5$ (resource requests leak) to violate the authorization goal (see Section II-C).

Said attacker assumption allows the attacker to obtain an access token at_{hon} from an honest flow, i.e., at_{hon} was issued by an honest AS AS_{hon} for an honest client c_{hon} on behalf of an honest user u (and thus at_{hon} is bound to keys of c_{hon}). By then injecting

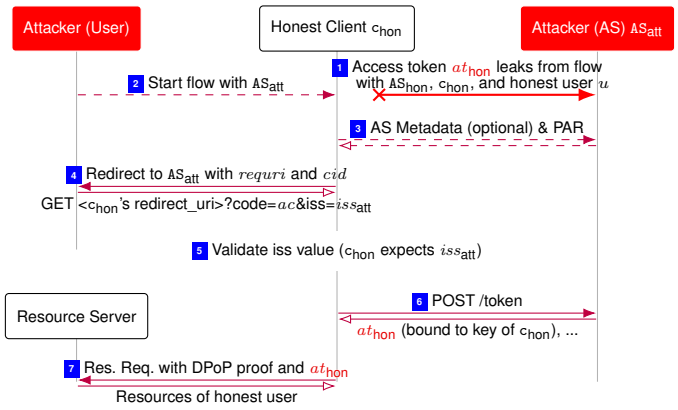

Figure 4. Cuckoo's token attack

at_{hon} into a flow between the attacker as user, c_{hon}, and an attacker-controlled AS, the attacker (as user of c_{hon}) gains access to u's resources.

Figure 4 shows the attack in detail: the attacker first acquires an access token at_{hon} from an honest flow (Step 1). Such a token may, for example, be obtained by the attacker through observing a resource request (see $\mathcal{A}5$). Due to access token sender constraining, the attacker cannot use at_{hon} directly at an RS (recall: at_{hon} is bound to keys of c_{hon}). Instead, the attacker, posing as a user towards c_{hon}, now starts a flow with c_{hon}, selecting an AS AS_{att} controlled by the attacker (Step 2). c_{hon} initiates the flow as usual with PAR (Step 3), followed by instructing the user, i.e., attacker, to visit AS_{att} (Step 4). Since AS_{att} is controlled by the attacker, the authentication and authorization steps can be skipped and the attacker (posing as a user) immediately "redirects" itself to c_{hon} with issuer identifier iss_{att} (which identifies AS_{att}) and an arbitrary authorization code ac (Step 4). Upon receiving that message, c_{hon} validates the issuer identifier, which succeeds: c_{hon} is and wants to be in a flow with AS_{att}. As usual, c_{hon} now sends a token request to AS_{att}, which responds with the previously acquired at_{hon} (Step 6). Recall that at_{hon} was issued for and is bound to keys of c_{hon} to access resources of u. Hence, c_{hon} can use at_{hon} at an honest RS. But since c_{hon} associates at_{hon} with the session between c_{hon} and the attacker posing as a user, this gives the attacker access to u's resources through c_{hon}.

At its core, this attack exploits the lack of binding between access token and AS from the client's point of view. Note: the client is mandated to handle the access token as an opaque value, i.e., cannot perform any checks on the token.

Fix. We proposed to fix this attack by mandating the client to include an AS issuer identifier in each resource request (which would be the attacker AS' identifier in the example above). The RS can then compare this issuer identifier sent by the client with the actual issuer of the access token (which, in our example, would be some different, honest AS). Note that in order to verify the token's validity, the RS already needs a way to know which AS originally issued the token. The FAPI WG acknowledged the attack [17] and added a description of the attack, as well as our proposed fix, to the specification [26]. However, the FAPI WG decided that the "preconditions for this attack do not apply to many ecosystems and require a powerful attacker" [32, Sec. 5.6.5] and hence made implementation of a fix optional. Note that while this fix seems to be a small change, there is no standardized way to send the issuer identifier of the AS, as well as no standardized way for the RS to get a value to compare against [32, Sec. 5.6.5]. Hence, mandating such a change would require substantial standardization efforts.

IV. Formal Analysis

In this section, we describe our formal analysis. We start with a primer on the WIM, continue with a description of our formal model of FAPI 2.0, including its limitations, and our formalization of a FAPI 2.0 Web System. We then discuss the most important differences and technical challenges of our work compared to prior work on authorization protocols using the WIM. This is followed by a description of the formalized security properties and a proof sketch for the authorization property; see the appendix for the full formal model and full proofs of all security properties.

A. The Web Infrastructure Model

FAPI 2.0 is a Web-based protocol, and the interaction between browsers and Web servers introduces potential attack surfaces, e.g., by cross-site requests, in-browser communication, malicious scripts, insecure headers, or redirections. To account for attacks originating from the browser and complex interactions inside browsers, as well as between parties, we analyze FAPI 2.0 based on the Web Infrastructure Model (WIM) [33], which is the most detailed formal model of the Web infrastructure to date. The WIM is a Dolev-Yao (DY) style pen-and-paper web model and requires manual analysis. It has successfully been applied to several web standards, to uncover previously unknown attacks and to prove security properties [21, 33, 34, 36–38] (see Section V). So far

no mechanized analysis framework has such a comprehensive model of the Web. Mechanizing such a very detailed model, from scratch or on top of existing tools, is a big challenge by itself and out of the scope of this work.

In the following, we give a high-level overview of the WIM closely following the summary in [36], with the full model given in Appendix G: the WIM is designed independently of a specific Web application and closely mimics published (de-facto) standards and specifications for the Web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM defines a general communication model, and, based on it, Web systems consisting of Web browsers, DNS servers, and Web servers as well as Web and network attackers.

Communication Model. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes that listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

As usual in DY models (see, e.g., [2]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the Web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, headers, and a message body. For example, a request for the URI http://example.com/s?p=1 is represented as

$$r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{example.com}, /\mathtt{s}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

where the body and the headers are empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}}))$ where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

The *equational theory* associated with Σ is defined as usual in DY models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example,

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

i.e., these two terms are equivalent w.r.t. the equational theory.

A *(DY) 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 computed (more formally, derived in the usual DY style) from the input event and the state.

The so-called *attacker process* is a DY process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any DY process could possibly perform. Attackers can corrupt other parties at any time; corrupted parties behave like the attacker process.

A *script* models JavaScript running in a browser. Scripts are defined similarly to DY processes. When triggered by a browser, a script is provided with state information, corresponding to the (browser) data available to JavaScript in real browsers. 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). Similarly to an attacker process, the so-called *attacker script* may output everything that is derivable from its input.

A *system* is a set of processes. A *configuration* (S, E, N) of this system consists of the states S of all processes in the system, the pool of waiting events E, and an infinite sequence of unused nonces N. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step. Such a transition is called *processing step* and denoted by

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

Here, the process p processes the event e_{in} and creates the output events E_{out} which are added to the pool of waiting events of the next configuration.

A *Web system* formalizes the Web infrastructure and Web applications. It contains a system consisting of honest and attacker processes. Honest processes can be Web browsers, Web servers, or DNS servers. Attackers can be either *Web attackers* (who can listen to and send messages from their own addresses only) or *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A Web system further contains a set of scripts (comprising honest scripts and the attacker script) and a mapping of these scripts to strings. A Web system also defines the pool of initial events, which typically only contains so-called trigger events, which trigger pre-defined actions (see below for an example for pre-defined browsers actions).

Web Browsers. An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the Web browser. User credentials are stored in the initial

state of the browser and are given to the respective Web pages, i.e., scripts. Besides user credentials, the state of a Web browser contains (among others) a tree of windows and documents, cookies, and Web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded Web page and contains (among others) a script and a list of subwindows (modeling iframes). The script, when triggered by the browser, is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and Web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XHRs and postMessages, submit forms, set/change cookies and Web storage data, and create iframes. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the Web standards.

A browser will typically send DNS and HTTP(S) requests as well as XHRs, and it processes the responses. Several HTTP(S) headers are modeled, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a trigger message upon which the browser non-deterministically chooses an action, for instance, to trigger a script in some document.

Generic HTTPS Server. The WIM defines a generic HTTPS server model which can be instantiated by application models. The generic server provides some generic functionality, e.g., a function for sending HTTPS requests, which internally handles DNS resolution and key management for symmetric transportation keys. The generic server also provides placeholder functions, e.g., for processing HTTPS requests and responses, which need to be instantiated by the application model.

B. Overview of FAPI 2.0 Model

We created the application-specific model in the WIM based on the FAPI 2.0 specifications. For this, we instantiated the generic HTTPS server model provided by the WIM to create detailed formal models for ASs, clients, and RSs, including models of scripts. Our formal model covers all essential mechanisms used by FAPI 2.0, e.g., AS Metadata, PAR, PKCE, the mTLS and `private_key_jwt` client authentication methods, DPoP and mTLS token sender constraining, and ID tokens. Our model also covers both structured and opaque access tokens—which can be different for each flow (the token type is chosen non-deterministically for each token request)—and token introspection.

Using client authentication and token sender constraining as examples, we exemplify how our model closely follows the specifications and show (parts of) some concrete messages within our model. We then continue with details on how we incorporated the attacker model and fixes described in Section III into our formal model, followed by a discussion of the limitations of our model. More details specific to the AS, client, and RS models are given in Appendix A and we provide the full formal models in Appendix D.

Client Authentication. Within the model, the ASs only accept PAR and token requests if they are client-authenticated, as mandated by the specifications [32, Sec. 5.3.1.2. No. 4], and clients always add client authentication when sending such requests [57, Sec. 2], [32, Sec. 5.3.2.1. No. 2]. Our model supports both mTLS and `private_key_jwt` authentication [32, Sec. 5.3.1.1. No. 6, Sec. 5.3.2.1. No. 2]. To illustrate how we model the `private_key_jwt` method, we show a client authentication JWT. Let cid be the client identifier of the client `c` at the AS `AS`, and let $iss_{\texttt{AS}}$ be the issuer identifier of `AS`. Within the model, $\mathsf{signkey}(\texttt{c})$ denotes a private signing key of `c`. With these values, a client authentication JWT is—closely following the specification [70, Sec. 9]—represented by the term $\mathsf{sig}([\texttt{iss}\colon cid, \texttt{sub}\colon cid, \texttt{aud}\colon iss_{\texttt{AS}}], \mathsf{signkey}(\texttt{c}))$, where `iss` is the issuer of the JWT, `sub` is the subject that is being authenticated, and `aud` is the audience value (i.e., the AS for which the JWT is being created).

Access Token Sender Constraining. Our AS model returns access tokens that are sender constrained by mTLS or DPoP [32, Sec. 5.3.1.1. No. 4, 5], and the client model sends the token request to the AS and the resource request to the RS with the corresponding proof-of-possession [32, Sec. 5.3.2.1. No. 1]. As mandated by FAPI 2.0, our model of RSs requires access tokens to be sender constrained using one of these methods [32, Sec. 5.3.3. No. 5]. We illustrate how we model a DPoP proof in a token request, where, according to the DPoP specification [39, Sec. 4.1], a client `c` adds a DPoP proof to the HTTP headers of its request: $headers[\texttt{DPoP}] := dpopProof$, where $dpopProof$ is a signed JWT as defined in [39, Sec. 4.2] (not to be confused with a client authentication JWT): $dpopProof := \mathsf{sig}(dpopJwt, \mathsf{signkey}(\texttt{c}))$ with $dpopJwt := [\texttt{headers}\colon [\texttt{jwk}\colon \mathsf{pub}(\mathsf{signkey}(\texttt{c}))], \texttt{payload}\colon [\texttt{htm}\colon \texttt{POST}, \texttt{htu}\colon tokenEndpoint]]$. The `jwk` value is the public key that the receiver can use for verifying the signature, the `htm` value is the HTTP method of the request in which the DPoP proof is included, e.g., `GET` or `POST`, and the `htu` value is the URL of that request (but with empty parameters and fragment).

1) Optional Fixes: As explained in Section III, the FAPI WG decided to make some fixes resulting from our attacks optional. In order to prove FAPI 2.0 secure, these fixes are required, hence our results only apply to implementations also employing these fixes. Here we describe how we deal with these fixes in the model.

Cuckoo's Token Attack. As described in Section III-E, the working group added (optional) countermeasures for preventing the attack. However, none of these fixes are mandatory, and each of them might mask other attacks. Thus, we decided to model a minimal fix that specifically targets this attack: right before sending the resource request to an RS, the client model checks whether the requested resource is managed by the AS from which it got the access token.

DPoP Proof Replay. For preventing this attack, we modeled replay protection using server-provided DPoP nonces (see also Section III-C). We require these nonces to be one-time use only, i.e., the RS model invalidates them after one use.

2) FAPI 2.0 Attacker Model: We initially modeled all attacker assumptions (see Section II-C), but as described in Section III, the FAPI WG decided to remove some of them in response to our analysis. We describe how we modeled the final assumptions (under which our proofs hold) and refer to Appendix C for the removed ones:

$\mathcal{A}1$ is part of the WIM, see Section IV-A.

$\mathcal{A}2$ (authorization requests leak) is modeled by leakage at the client: After creating the authorization request URI, the client can non-deterministically decide whether to leak it. In the leak case, it non-deterministically chooses an IP address and sends out the request to this IP (in plain). Thus, the client model leaks the client identifier and the request URI value of the authorization request.

$\mathcal{A}5$ (resource requests leak) was initially modeled by leaking the resource request at the RS after receiving the request (and in the case of opaque tokens, before receiving the introspection response). As described in Section III-C, this enables the attacker to replay DPoP proofs. In addition to using server-provided DPoP nonces, we adapted the model (in line with changes to the FAPI 2.0 Attacker Model, see Section III-C) such that the resource request leaks *after* the RS responds with the resource response (i.e., after invalidating the DPoP nonce).

3) Limitations of Our Model: While our model covers most of the FAPI 2.0 specifications, there are a few things which we handle on a very abstract level or do not model at all (besides the inherent abstractions of the WIM, such as details of TLS).

Error Handling. FAPI 2.0 and several of the underlying specifications define a set of error messages, e.g., when client authentication fails. These are not represented in our model: if a process encounters an error condition, it just aborts the current processing step without output and without changes to the process state. We note that none of the specifications mandate a certain behavior upon *receiving* an error message.

Rich Authorization Requests. *Rich Authorization Requests (RAR)* [56] is a mechanism which allows clients to be more precise about what kind of access they request. However, FAPI 2.0 does not mandate the use of RAR, and RAR semantics are subject to individual AS policies. Hence, RAR does not allow for a general treatment, and hence, is not covered by our model.

Modeled Grant Types. The OAuth 2.0 framework [41], as well as OpenID Connect [70], define various grant types, such as the implicit grant, the hybrid grant, and the authorization code grant. FAPI 2.0 explicitly excludes all of them, except for the client credentials grant and the authorization code grant, where only the latter is required to be supported by FAPI 2.0 ASs and clients. In the client credentials grant, the client takes the role of the user, hence, removing the additional interaction between user and client. So, this grant is subsumed by the authorization code grant, which is why we only model and analyze the latter.

Refresh Tokens. Even though FAPI 2.0 allows for refresh tokens (see [41]) to be used, we do not model them. Instead, we model regular access (and id) tokens as having an indefinite lifetime, i.e., they never expire.

Native Clients with Loopback Redirect. FAPI 2.0 ASs must reject PAR requests with a redirect uri using the http scheme, i.e., where the client's redirection endpoint is not protected by TLS (cf. Step 5 in Figure 1). There is, however, one exception to this rule: if the client is a native client running on the same device as the browser and is using loopback interface (i.e., device-local) redirection, it may use an http redirect uri (see [18] and [31, Sec. 4.3.1.2 No. 8]). In our AS model, we do not allow http redirect uris.

C. Comparison to WIM Analyses of Related Protocols

We here describe specific differences between our work and previous WIM analyses of FAPI 1.0 [38], OAuth 2.0 [36], and OpenID Connect [37].

Since FAPI 2.0 is a completely redesigned protocol, obviously, previous analyses do not apply to FAPI 2.0. The differences are also apparent by the new vulnerabilities and attacks we have found on FAPI 2.0.

While FAPI 2.0 is based on OAuth 2.0 and OpenID Connect, it not only contains many additional mechanisms, such as PAR, DPoP, mTLS, PKCE, Authorization Server Issuer Identification, AS metadata and so on, but also aims to be secure under a much stronger attacker, see also below.

Similarly, while FAPI 1.0, the predecessor of FAPI 2.0, was formally analyzed [38], FAPI 2.0 is a very different protocol: in comparison to FAPI 1.0, several security mechanisms have been removed, most notably *OAuth Token Binding* [43] and *JWT Secured Authorization Response Mode (JARM)* [53]. On the other hand, FAPI 2.0 adds new mechanisms, such as DPoP and PAR, which have not undergone any formal treatment so far. Also, FAPI 2.0 mandates client authentication, removes support for public clients as well as the hybrid flow, and introduces a stronger attacker model.

In the following, we briefly discuss further details and differences.

Token Introspection. Previous work on OAuth 2.0 and OpenID Connect does not consider the resource server, and while prior work on FAPI 1.0 does, token introspection is modeled only on a very abstract level: instead of querying the AS, the RS uses an idealized introspection oracle. In contrast, our AS and RS models contain a detailed representation of token introspection as defined in RFC 7662 [69]. Hence, we have to reason about additional messages, their contents, as well as integrity, authentication, and secrecy thereof.

Structured Access Tokens. Prior work on OAuth 2.0, OpenID Connect, and FAPI 1.0 does not contain a model of structured access tokens and their use, in particular at the RS. Note that besides the additional options for flows introduced by structured access tokens, modeling them also requires additional keys to be distributed.

Number of ASs Supported by an RS. As mentioned above, prior work on OAuth 2.0 and OpenID Connect did not model the RS. In the FAPI 1.0 analysis in [38], the RS supports one fixed AS, whereas our RS model supports an arbitrary set of ASs, some of which may be corrupted. Hence, our analysis considers a broader set of cases and possible mix-ups.

RS Authentication. RFC 7662 [69] mandates for RSs to authenticate to the AS during token introspection. Since previous work on OAuth 2.0, OpenID Connect, and FAPI 1.0 did not model token introspection, our model is the first (not just among WIM analyses) incorporating RS authentication. Similar to structured access tokens above, this requires distribution of, and reasoning about, additional secrets.

DPoP Access Token Sender Constraining. The DPoP [39] mechanism has only recently been standardized and hence, has not been part of any formal analysis to the best of our knowledge. With its additional signatures and subtle details regarding the signed data, in particular, the exact contents of the `htu` value (see [39, Sec. 4.2]), DPoP requires careful modeling of the AS, the RS, and the client. In addition, DPoP requires key material to be distributed between client, AS, and RS (as well as additions to the token introspection and structured access token models).

PAR. Similar to DPoP above, PAR [57] has only recently been standardized and has thus not been part of any formal analysis. While PAR does not require additional key material, it adds more options and messages to the FAPI 2.0 flows, hence necessitating reasoning about them.

JWT Client Authentication. While the `public_key_jwt` client authentication method has been standardized as part of OpenID Connect [70], it has not received any formal security analysis so far, including WIM analyses of OpenID Connect and FAPI 1.0 (FAPI 1.0 is based on OpenID Connect and explicitly allows for `public_key_jwt` client authentication). Hence, our work is the first to consider this kind of client authentication.

Attacker Model. Compared to prior WIM work on OAuth 2.0, OpenID Connect, and FAPI 1.0, our analysis considers a significantly stronger attacker model. Where [36] and [37] consider a fairly standard Dolev-Yao attacker (i.e., similar to $\mathcal{A}1$, see Section II-C), [38] considers a stronger attacker: the attacker in [38] is a network attacker similar to our $\mathcal{A}1$, which also may access authorization requests (cf. $\mathcal{A}2$) and authorization responses (cf. $\mathcal{A}3$) in plain, as well as force honest clients to use attacker-controlled token endpoints (cf. $\mathcal{A}4$). In addition, the authors of [38] consider access tokens leaking to the attacker, which is something that is subsumed by our attacker assumption $\mathcal{A}5$ (resource request leak). However, our attacker model is even stronger in that (1) $\mathcal{A}5$ not only leaks the access token, but also all other values in the resource request, in particular DPoP proofs, (2) $\mathcal{A}6$ leaks resource responses, and (3) $\mathcal{A}7$ even allows for the attacker to modify resource responses. To the best of our knowledge, neither (1), nor (2), nor (3) have previously been considered in any formal analysis of authorization protocols.

Session Integrity Properties. While prior works on OAuth 2.0 and OpenID Connect do consider session integrity properties similar to ours, these are only proven in a setting with *web attackers*, i.e., the attacker may corrupt any party, but does not control the network and cannot spoof sender addresses. Similar properties have also been proven for FAPI 1.0 *with* a network attacker, however, their property requires the use of *OAuth 2.0 Token Binding* [43], a mechanism to bind authorization codes and access tokens to TLS connections, which is quite different from DPoP and mTLS access token sender constraining.

D. FAPI 2.0 Web System

As outlined in Section IV-A, a web system formalizes the overall system covered by our analysis. We call $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ a *FAPI 2.0 web system with network attacker*, or short $\mathcal{FAPI}$, if the components of the web system are defined as follows:

$\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ 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 $\mathsf{Hon} := \mathsf{B} \cup \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$. The honest processes behave as outlined in Section IV-A and Section IV-B (formally specified in Appendix D). DNS servers are subsumed by the network attacker and are therefore not modeled explicitly.
$\mathcal{S}$ contains the attacker script R^{att} outlined in Section IV-A (formally specified in Definition 55), the script $script\_as\_form$ for logging in the resource owner at the AS, and the script $script\_client\_index$ for selecting an AS and initiating a flow at the client as outlined in Appendix A (both scripts are formally specified in Appendix D). The mapping script maps these scripts to strings: $script\_client\_index \mapsto$ `script_client_index`, $script\_as\_form \mapsto$ `script_as_form`, $R^{\text{att}} \mapsto$ `att_script`.
E^0 is the initial set of events and contains only trigger events (see Section IV-A).

Note that we prove security properties about all such FAPI 2.0 web systems with network attacker, and hence, systems with an arbitrary number of browsers, clients, ASs, and RSs, and in which each party can have an arbitrary number of parallel, interleaving protocol sessions. Also recall from Section IV-A that the attacker can corrupt honest parties at any time.

E. Formal Security Properties

As described in Section II-C, the FAPI WG wants the protocol to meet authorization, authentication, and session integrity goals. In the following, we present our formalized authorization and authentication properties, capturing the corresponding security goals. We present the formalized session integrity properties in Appendix E. We conclude this section with our main security theorem.

1) Authorization: Recall that informally, authorization means that an attacker should never be able to access resources of honest users (unless the user authorized such access). We highlight that this statement covers many different scenarios, for example, that the attacker cannot use leaked access tokens at the RS and cannot, by some mix-up, force an honest client to use an access token associated with an honest user in a session with the attacker.

We first formalize that an access token was issued by AS $\texttt{AS}$, is bound to a public key k, and is associated with a user identity id, and give our authorization property afterwards.

Definition 1 (Access Token bound to Key, AS and ID). Let $\mathcal{FAPI}$ be a FAPI 2.0 web system with network attacker, $k \in \mathcal{T}_{\mathcal{N}}$ be a term, $\texttt{AS} \in \mathsf{AS}$ an AS, and $id \in \mathsf{ID}$ a user identity.[2] We say that a term t is an *access token bound to* k, $\texttt{AS}$, *and* id in configuration (S, E, N) of a run of $\mathcal{FAPI}$, if there is an entry $rec \in^{\langle\rangle} S(\texttt{AS}).\texttt{records}$ (i.e., in the state of $\texttt{AS}$) such that $rec[\texttt{access\_token}] \equiv t$ and $rec[\texttt{subject}] \equiv id$ and $(rec[\texttt{cnf}] \equiv [\texttt{jkt}\colon \mathsf{hash}(k)]) \vee (rec[\texttt{cnf}] \equiv [\texttt{x5t\#S256}\colon \mathsf{hash}(k)])$.
Informally, this means that t is stored in the state of $\texttt{AS}$, together with the identity id and the hash of k. If the key is stored under the name $\texttt{jkt}$, then the token is bound via DPoP, otherwise, it is bound via mTLS.

The following authorization property captures the following: if an honest RS rs provides access to a resource r of an honest resource owner with user identity id managed by an honest AS $\texttt{AS}$, then the following holds true: (I) rs has received a request for accessing the resource r with an access token at in the same (which is possible if the token at is structured and can be verified by the RS immediately) or in a previous processing step (if the token at is opaque to the RS and it thus performed token introspection), and rs created the resource when receiving the resource request (see Appendix A-C on how our model manages resources). (II) The token at is bound to some key k, $\texttt{AS}$, and the user identity id (see Definition 1). (III) If k is the key of an honest client, then the attacker cannot derive the resource.

Definition 2 (Authorization Property). We say that a FAPI 2.0 web system with 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 RS $rs \in \mathsf{RS}$ 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 $Q = (S^Q, E^Q, N^Q) \xrightarrow[rs \to E^Q_{\text{out}}]{e^Q_{\text{in}} \to rs} (S^{Q\prime}, E^{Q\prime}, N^{Q\prime})$ in ρ, every $resourceID \in \mathbb{S}$ with $\texttt{AS} = \mathsf{authorizationServerOfResource}^{rs}(resourceID)$ being honest in S^Q, it holds true that:[3]

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

(I) $\exists$ a processing step $P = s_i \xrightarrow[rs \to E^P_{\text{out}}]{e^P_{\text{in}} \to rs} s_{i+1}$ such that
  a) either $P = Q$, or P is 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_{req} is an HTTP request which contains a term (access token) t in its $\texttt{Authorization}$ header, i.e., $t \equiv m_{\text{req}}.\texttt{headers}[\texttt{Authorization}].2$, and
  c) r is a fresh nonce generated in P at the resource endpoint of rs in Line 46 of Algorithm 12.

(II) t is bound to a key $k \in \mathcal{T}_{\mathcal{N}}$, $\texttt{AS}$, and id in S^Q (see Definition 1).

(III) 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., $r \notin d_\emptyset(S^n(\mathsf{attacker}))$.

2) Authentication: Recall that the authentication goal states that an attacker should not be able to log in at an honest client under the identity of an honest user. In our model, the client sets a cookie that we call *service session id* at the browser after a successful login. The client model stores the service session id in its $\texttt{sessions}$ state subterm, and associates with it the identity that is logged in to the session (the identity is taken from an ID token, see Appendix A-B for more details on the login of a user at a client). On a high level, our formalized property states that an attacker should not be able to derive the service session id for a session at an honest client where an honest identity is logged in, as long as the identity is managed by an honest AS. We stress that this not only covers that a cookie set at the browser of the honest user does not leak, but that there is no way in which the attacker can log in at an honest client as an honest user.

[2] ID is a set of terms of the form $\langle name, domain\rangle$, where $name$ is a string, the user name, and $domain$ is a domain (usually of an AS). We also define a mapping $\mathsf{ownerOfID}\colon \mathsf{ID} \to \mathsf{B}$ which maps an identity to the browser whose user owns the identify (users are modeled as part of their browser). Likewise, we define a mapping $\mathsf{governor}\colon \mathsf{ID} \to \mathsf{AS}$ mapping identities to "their" AS. See Appendix D-B and Appendix D-D for details.

[3] $\mathsf{authorizationServerOfResource}^{rs}$ is a mapping from resource ids to the authorization server that manages the respective resource, see Definition 10.

We start with an auxiliary definition, capturing that the client logged in a user with a service session id, before presenting the authentication property itself.

Definition 3 (Service Sessions). We say that there is a *service session identified by a nonce n for a user identity id at some client* c in a configuration (S, E, N) of a run ρ of a FAPI 2.0 web system with network attacker $\mathcal{FAPI}$ iff there exists some session id x and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that

$$S(\mathsf{c}).\mathtt{sessions}[x][\mathtt{loggedInAs}] \equiv \langle d, id \rangle$$

and

$$S(\mathsf{c}).\mathtt{sessions}[x][\mathtt{serviceSessionId}] \equiv n.$$

Definition 4 (Authentication Property). We say that a FAPI 2.0 web system with 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 $\mathsf{c} \in \mathsf{C}$ that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathtt{AS} = \mathsf{governor}(id)$ 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., $n \notin d_\emptyset(S(\mathtt{attacker}))$).

3) Security Theorem: As described in Section II-C, the protocol aims to fulfill authorization, authentication, and session integrity properties. Thus, our overall security theorem is the conjunction of all four properties, where as mentioned session integrity for authorization and authentication is defined in Appendix E.

Theorem 1. Every FAPI 2.0 web system with network attacker $\mathcal{FAPI}$ fulfills authorization, authentication, session integrity for authentication, and session integrity for authorization.

We highlight that we prove this theorem for the powerful attacker described in Section IV-B2 within a faithful formal model that includes the fixes described in Section III. We also emphasize that our analysis takes into account many Web features that can be the root of attacks: e.g., the browser model allows for the execution of scripts loaded from different websites/origins at the same time, possibly with malicious scripts. The model also considers fine-grained behavior of HTTP redirects,[4] several security-critical headers, as well as subtleties of various cookie attributes, which, for example, could result in vulnerable session management, and in-browser communication using postMessages, just to name a few of the Web features considered in our analysis. Thus, our analysis excludes attacks that arise from features of the Web infrastructure. Our proof of Theorem 1, which, due to space limitations, we give in Appendix F, consists of more than 20 lemmas and of course reasons about the full formal model that we provide in Appendix D.

F. Proof Sketch for Authorization

In this section, we summarize our proof of the authorization property. We refer to Appendix F for the full proof of this as well as the authentication and session integrity properties.

Recall that on a high level, the authorization property requires that the attacker cannot access resources of an honest identity id stored at an honest RS.

For the **first postcondition**, see Definition 2 (I), we show that whenever the RS gives access to a resource of an identity (the resource owner), it does so in response to a request that contains an access token. We show this for both types of access tokens, i.e., for structured tokens signed by the AS managing the resource, as well as opaque tokens that the RS has to verify with the AS via token introspection. Unlike the other two, this postcondition is quite easy to show.

For the **second postcondition**, see Definition 2 (II), we show that the access token was issued by the AS managing the resource and is bound to some key k and the identity id of the resource owner (see Definition 1). Keep in mind that Definition 1 refers to the state of AS. For structured access tokens, we show this by reasoning on the validity of the token: The RS checks that the token has a valid signature by the AS managing the resource, i.e., the token was indeed created by that AS (we give more details in the full proof, e.g., proving that the signing key of AS is secret). We show that the AS, when creating the token, stores the token in its state, together with id and k that it also puts into the structured token (i.e., binds the token to k and id). For opaque tokens, the introspection endpoint of AS returns the identity id and the key k (the RS only provides access in case of a successful introspection response). The AS only sends such a response if it previously bound the token to k and id.

The **third postcondition**, see Definition 2 (III), states that if the key k is the private (TLS or DPoP) key of an honest client c, then the attacker does not get access to the resource. Showing this property is more involved than the previous ones.

We first show that the resource request was indeed sent by c: If the RS received a DPoP proof (for the key that the token is bound to), then we can apply Lemma 12 which gives us that the DPoP proof can only be derived by c (and rs, which just received it). Note that this only holds true with the fix that we propose in Section III-C (i.e., effective replay protection) and with the updated $\mathcal{A}5$ attacker assumption (see Section IV-B2). Similarly, we show the effectiveness of mTLS token binding, i.e., that access tokens bound to a TLS key of an honest client can only be used by that client. Overall, we show for both structured and opaque access

[4]For example, FAPI 2.0 excludes code 307 redirects, as they would cause attacks similar to [36]

tokens that the resource request must have been sent by c. Thus, the RS provides c access to the resource of id (recall that all communication between parties happens via secure channels).

We then prove that c does not provide the attacker access to the resource, i.e., the attacker cannot use c (e.g., as a user) to access resources of id. We show this by contradiction, i.e., we assume that c provides the attacker access, and lead this to a contradiction. In the full proof, we show that this can only happen if the redirection request (Step 12 in Figure 1) was sent by the attacker (recall that c expects a request to its redirection endpoint with an authorization code $code$ that c then uses at the token endpoint of the AS). We then show that c sends the token request to the correct AS, i.e., to the one that is managing the resource. This is true thanks to the fix for preventing the Cuckoo's Token attack (see Section IV-B1). As the access token that c receives from the token endpoint provides access to resources of id, it follows that $code$ is also associated with id (i.e., the AS stored in its state that $code$ was issued when id logged in at its authorization endpoint). Lemma 5 now gives us that c received this exact $code$ at its redirection endpoint. As the request to the redirection endpoint is assumed to originate from the attacker, we have that the attacker can derive $code$. However, this is a contradiction to Lemma 17, which states that the authorization code of a flow between an honest AS and honest client is not derivable by any other process, including the attacker. This concludes the proof.

V. Related Work

While there has not been much work on the relatively recent and partly still under development FAPI standards and specifications so far, standards like OAuth 2.0 and OpenID Connect, on which FAPI is based, have received quite some attention by security researchers. Besides many studies on implementations and deployments [77–79, 82], OAuth 2.0 has been formally analyzed: Pai et al. [65] built a limited model of OAuth 2.0, lacking many generic web features, for the Alloy finite-state model checker and showed that with their approach, known weaknesses can be found. Chari et al. [14] analyzed the authorization code flow in the UC model and found no attacks, but their model omits many web features. Two more comprehensive formal analyses have been conducted by Bansal et al. [5, 6], using their WebSpi library and ProVerif, in which they modeled various settings of OAuth 2.0, e.g., with CSRF vulnerabilities in ASs and clients. Bansal et al. uncovered several previously unknown attacks on various popular OAuth 2.0 implementations. However, their works main focus was on finding attacks, rather than proving security. The latter was the focus of a formal analysis by Fett et al. [36], in which, despite prior formal analysis efforts, they discovered several new attacks, and were only able to prove security after developing fixes for them. As in our work, the analysis by Fett et al. is based on the WIM, i.e., includes a comprehensive formal model of the web infrastructure. Likewise, OpenID Connect has been analyzed before [37, 52, 58].

However, as described in more detail in Section IV-C, FAPI 2.0 is a completely redesigned protocol, and hence, previous analyses on related protocols do not apply. Finally, as already mentioned, FAPI 2.0 itself has not undergone any formal security analysis so far.

VI. Conclusion

Asked by the OpenID Foundation's FAPI working group, we accompanied the development of the FAPI 2.0 specifications with a formal security analysis, by creating formal models and formalizing security properties closely following the FAPI 2.0 specifications, including their strong attacker model. During this formal analysis, we discovered several attacks that violate the security goals set by the FAPI 2.0 specifications. These have been reported to the FAPI WG, who acknowledged our attacks. We then worked with them to incorporate many of our proposed fixes and improvements into the official specifications. This finally allowed us to provide formal security proofs of all security properties.

Performing such an analysis in a meaningful model like the WIM ensures that attacks based on many threats and features of the web and their complex interactions can be found and ruled out. We note that our analysis is a pen-and-paper analysis, as current tools do not directly support models that are as detailed as the WIM. Hence, our proofs are not mechanized and, given the complexity of the protocol, are inherently lengthy, hence, tedious to verify. Nevertheless, our formal and systematic analysis has helped improve the standard and gain more insights and confidence in its security. Conversely, we consider mechanized analyses of web protocols as interesting future work.

Surprisingly, besides new vulnerabilities and attacks, we also uncovered some attacks which were similar to known attacks on related protocols, and in particular known within the FAPI WG. These findings further underline the importance of a systematic, formal analysis, as for complex protocols, like FAPI 2.0, also experts easily overlook even known attack patterns.

Overall, our contributions to the standardization process were very welcomed by the FAPI WG and led to significant improvements of an important protocol just in time: FAPI 2.0 is about to be adopted in highly sensitive environments, with millions of users managing bank account data, financial transactions, eGovernment applications, and even health data.

Acknowledgment

We thank our anonymous reviewers for their invaluable feedback. This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 443324941, and by the OpenID Foundation.

References

[1] *A05:2021 – Security Misconfiguration*. URL: owasp.org/Top10/A05_2021-Security_Misconfiguration/.

[2] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *ACM POPL*. 2001, pp. 104–115.

[3] *Authorization Request Leaks lead to CSRF*. URL: bitbucket.org/openid/fapi/issues/534.

[4] *Bank Account Aggregation Services Australia*. URL: www.creditsense.com.au/bank-account-aggregation/.

[5] C. Bansal et al. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *IEEE CSF*. 2012.

[6] C. Bansal et al. "Discovering concrete attacks on website authorization by formal analysis". In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

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

[8] V. Bertocci. *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. RFC 9068.

[9] *Browser swap attack explained on 2022-09-28*. URL: bitbucket.org/openid/fapi/issues/543.

[10] *Budget Tracker & Planner*. URL: mint.intuit.com/.

[11] B. Campbell et al. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705.

[12] *Car API platform for connected vehicle data*. URL: smartcar.com/.

[13] *Change attacker model to reflect formal model*. URL: bitbucket.org/openid/fapi/pull-requests/381.

[14] S. Chari et al. *Universally Composable Security Analysis of OAuth v2.0*. Cryptology ePrint Archive, Paper 2011/526. 2011. URL: eprint.iacr.org/2011/526.

[15] L. Chen et al. *Cookies: HTTP State Management Mechanism*. IETF Draft draft-ietf-httpbis-rfc6265bis-09. 2021.

[16] *Consumer Data Standards*. URL: consumerdatastandards.gov.au/.

[17] *Decide on what to do for A. Cuckoo's Token Attack*. URL: bitbucket.org/openid/fapi/issues/525.

[18] W. Denniss and J. Bradley. *OAuth 2.0 for Native Apps*. RFC 8252.

[19] *Die elektronische Patientenakte*. URL: www.bundesgesundheitsministerium.de/elektronische-patientenakte.html.

[20] *Discovery should be mandated for clients*. URL: bitbucket.org/openid/fapi/issues/536.

[21] Q. H. Do et al. "A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification". In: *IEEE S&P*. 2022, pp. 215–234.

[22] *DPoP & resource leaks*. URL: bitbucket.org/openid/fapi/issues/533.

[23] *Electronic Prescription Service - FHIR API*. URL: digital.nhs.uk/developer/api-catalogue/electronic-prescription-service-fhir.

[24] *email – Graph API*. URL: developers.facebook.com/docs/permissions/reference/email.

[25] *FAPI2SP: Add requirement for RP to use discovery*. URL: bitbucket.org/openid/fapi/pull-requests/363.

[26] *FAPI2SP: Add security consideration for cuckoo's token attack*. URL: bitbucket.org/openid/fapi/pull-requests/364.

[27] *FDX*. URL: financialdataexchange.org/.

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

[29] D. Fett. *FAPI 2.0 Attacker Model. 2nd Implementer's Draft*. 2022. URL: openid.net/specs/fapi-2_0-attacker-model-02.html.

[30] D. Fett. *FAPI 2.0 Attacker Model, Commit 209f58a*. OpenID Foundation. 2022. URL: bitbucket.org/openid/fapi/src/209f58a/FAPI_2_0_Attacker_Model.md.

[31] D. Fett. *FAPI 2.0 Baseline Profile, Commit 209f58a*. OpenID Foundation. 2022. URL: bitbucket.org/openid/fapi/src/209f58a/FAPI_2_0_Baseline_Profile.md.

[32] D. Fett. *FAPI 2.0 Security Profile. 2nd Implementer's Draft*. 2022. URL: openid.net/specs/fapi-2_0-security-02.html.

[33] D. Fett et al. "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System". In: *IEEE S&P*. 2014, pp. 673–688.

[34] D. Fett et al. "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web". In: *ESORICS*. 2015, pp. 43–65.

[35] D. Fett et al. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *ACM CCS*. 2015.

[36] D. Fett et al. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *ACM CCS*. 2016.

[37] D. Fett et al. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *IEEE CSF*. 2017.

[38] D. Fett et al. "An Extensive Formal Security Analysis of the OpenID Financial-grade API". In: *IEEE S&P*. 2019, pp. 1054–1072.

[39] D. Fett et al. *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)*. IETF Draft draft-ietf-oauth-dpop-08. 2022.

[40] *Governance of NZ payment systems*. URL: www.paymentsnz.co.nz/.

[41] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749.

[42] *Improve attacker model description after introduction of metadata*. URL: bitbucket.org/openid/fapi/pull-requests/371.

[43] M. Jones et al. *OAuth 2.0 Token Binding*. IETF Draft draft-ietf-oauth-token-binding-08. 2018.

[44] M. Jones et al. *Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7521.
[45] M. Jones et al. *JSON Web Signature (JWS)*. RFC 7515.
[46] M. Jones et al. *JSON Web Token (JWT)*. RFC 7519.
[47] M. Jones et al. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7523.
[48] M. Jones et al. *OAuth 2.0 Authorization Server Metadata*. RFC 8414.
[49] A. Kumar. "Using automated model analysis for reasoning about security of web protocols". In: *ACM ACSAC*. 2012.
[50] A. Kumar. "A Lightweight Formal Approach for Analyzing Security of Web Protocols". In: *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 192–211.
[51] R. Küsters et al. *The Web Infrastructure Model (WIM)*. Technical Report. Version 1.0. 2022. URL: www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf.
[52] W. Li and C. J. Mitchell. "Analysing the Security of Google's Implementation of OpenID Connect". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 357–376.
[53] T. Lodderstedt and B. Campbell. *Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)*. OpenID Foundation. 2018. URL: openid.net/specs/openid-financial-api-jarm.html.
[54] T. Lodderstedt et al. *IETF 105 Materials: OAuth 2.0 Security Best Current Practice*. 2019. URL: datatracker.ietf.org/meeting/105/materials/slides-105-oauth-sessa-oauth-security-topics-00.pdf.
[55] T. Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. IETF Draft ietf-oauth-security-topics-19. 2021.
[56] T. Lodderstedt et al. *OAuth 2.0 Rich Authorization Requests*. IETF Draft draft-ietf-oauth-rar-15. 2022.
[57] T. Lodderstedt et al. *OAuth 2.0 Pushed Authorization Requests*. RFC 9126.
[58] V. Mladenov et al. "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect". In: *CoRR* abs/1508.04324v2 (2015).
[59] *OAuth 2.0 Scopes for Google APIs*. URL: developers.google.com/identity/protocols/oauth2/scopes#drive.
[60] *OAuth 2.0 Simplified: Token Introspection Endpoint*. URL: www.oauth.com/oauth2-servers/token-introspection-endpoint/.
[61] *OAuth Guide*. URL: developers.dropbox.com/oauth-guide.
[62] *Open Banking UK*. URL: www.openbanking.org.uk/.
[63] *Open Finance*. URL: www.bcb.gov.br/en/financialstability/open_finance.
[64] *Open Finance*. URL: www.gov.br/susep/pt-br/assuntos/open-insurance.
[65] S. Pai et al. "Formal Verification of OAuth 2.0 Using Alloy Framework". In: *IEEE CSNT*. 2011.
[66] *Reduced attacker model*. URL: bitbucket.org/openid/fapi/pull-requests/377.
[67] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617.
[68] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246.
[69] J. Richer. *OAuth 2.0 Token Introspection*. RFC 7662.
[70] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. 2014. URL: openid.net/specs/openid-connect-core-1_0.html.
[71] N. Sakimura et al. *OpenID Connect Discovery 1.0 incorporating errata set 1*. OpenID Foundation. 2014. URL: openid.net/specs/openid-connect-discovery-1_0.html.
[72] N. Sakimura et al. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636.
[73] N. Sakimura et al. *The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)*. RFC 9101.
[74] *Scopes for OAuth*. URL: docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps.
[75] K. M. zu Selhausen and D. Fett. *OAuth 2.0 Authorization Server Issuer Identification*. RFC 9207.
[76] *Services provided at Helsenorge*. URL: www.helsenorge.no/en/about-services-on-helsenorge/how-to-use-helsenorge-services#services-provided-at-helsenorge.
[77] E. Shernan et al. "More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 239–260.
[78] S.-T. Sun and K. Beznosov. "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems". In: *ACM CCS*. 2012.
[79] R. Yang et al. "Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations". In: *ASIA CCS*. 2016.
[80] *yes.com*. URL: yes.com/.
[81] *YouTube Data API Overview*. URL: developers.google.com/youtube/v3/getting-started.
[82] Y. Zhou and D. Evans. "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities". In: *USENIX*. 2014, pp. 495–510.

APPENDIX A
DETAILS ON APPLICATION-SPECIFIC MODEL

In this appendix, we continue the overview of our formal model from Section IV-B with additional details and references to the respective specifications, demonstrating how our model closely follows the specifications.

A. Authorization Server Model

Pushed Authorization Request. We model the PAR endpoint as part of the AS (mandated by FAPI 2.0 [32, Section 5.3.1.2. No. 2]). The endpoint model stores all values and returns a (freshly chosen) request URI value. In particular, as mandated by FAPI 2.0, the model requires a redirect URI and a PKCE code challenge value in the PAR request [32, Section 5.3.1.2. No. 5, 6].

Token Introspection. The introspection endpoint part of the AS model expects an opaque access token and returns whether the token is active (and thus, if the token was issued by this AS at all), information on the key to which the token is bound, and a subject identifier, i.e., the identity of the user whose login at the AS lead to the AS issuing the token. As required in [69, Section 2.1], the AS model requires successful authentication of the RS that sent the request. The exact authentication mechanism for RSs at ASs is out of scope for FAPI 2.0. In our model, we use the HTTP Basic Authentication mechanism suggested by [69].

Login Script. Upon receiving the authorization request, the AS model responds with a script that models the login page for the user. As a result of executing the script, the browser sends a POST request to the AS with the login credentials (for the current AS) of a user that is using the browser.

Further Endpoints. The AS model also comprises an endpoint for server metadata that returns information about the server such as the different endpoint URLs. This endpoint is mandated by FAPI 2.0 [32, Section 5.3.1.1. No. 1]. The authorization endpoint of the AS model requires a request URI value as mandated in [32, Section 5.3.1.2. No. 3]. Furthermore, the model has a JWKS endpoint, where the server responds with the public signature verification key, which is strongly recommended by FAPI 2.0 [32, Section 5.6.3].

B. Client Model

Configurations. As noted above, the client authenticates either via mTLS or the `private_key_jwt` method, and supports sender constraining by either mTLS or DPoP. In the model, the client can have different combinations of client authentication and sender constraining methods for different ASs. However, for a given pair of client and AS, this configuration does not change, reflecting that clients must register with ASs before starting a flow and select a configuration as part of that registration. We note that this registration is out of scope for FAPI 2.0 and we model it as part of the initial states of clients and ASs.

Starting a Flow. For starting a flow, the client model provides a script for browsers which triggers a POST request to the client. In our model, this request must contain the domain of an AS, modeling a user selecting an AS. When starting a flow for the first time with an AS, the client fetches the server metadata from the AS (as required by FAPI 2.0 [32, Section 5.3.2.1. No. 9]).

End-User Authentication. For each flow, the client decides non-deterministically whether it wants to authenticate the user. If that is the case, the client adds the value `openid` to the `scope` value contained in the pushed authorization request, hence requesting an id token from the AS. If the client model requests an id token, and once it receives the token response, it non-deterministically decides whether to log the user in based on the id token or to redeem the access token. The client logs in the user by creating a fresh cookie (called *service session id* in the client model) associated with the subject identifier contained in the id token.

Further Client Aspects. Furthermore, as FAPI 2.0 mandates, the client model always uses pushed authorization requests and PKCE, and always checks the iss issuer identifier in the redirection request (at the redirection endpoint) [32, Section 5.3.2.2. No. 2, 3, 4].

Another important detail concerns the handling of id tokens: FAPI 2.0 uses OpenID Connect id tokens, which means that id tokens are signed by their issuer. This of course raises the question of why the honest client in the attacker token injection attack accepts an attacker-constructed id token (see Section III-A). The reason is that OpenID Connect [70, Sec. 3.1.3.7 No. 6] (and thus FAPI 2.0) allows clients to skip signature verification on id tokens if the token is received directly from the token endpoint over a TLS-protected connection – these conditions are always fulfilled with FAPI 2.0. In the attacker token injection attack, the client contacts an attacker-controlled token endpoint (via a TLS-protected connection) and thus does not verify the id token signature.

C. Resource Server Model

Verification of Access Token. A request for a resource must contain an access token in the HTTP headers [32, Section 5.3.3 No. 1]. The RS model verifies that the token is valid, identifies the resource owner for whose resources the token was issued, and the key to which the token is bound as follows. If the token is a structured token, the RS checks the signature of the token using the public verification key of the AS responsible for the requested resource and retrieves the resource owner information, as well as the key to which the token is bound from the token. Otherwise, the RS model sends the token and RS authentication information to the introspection endpoint of the AS that manages the requested resource. The token introspection response then contains the necessary information.

Modeling Resources. As the management of resources is not within the scope of the FAPI 2.0 specifications, we assumed a generic resource management which we describe in the following: The RS model manages different resources identified by URLs, in particular, by the path part of URLs. For a given resource, the RS model knows which AS manages the resource. Note that this is common (and necessary) in real ecosystems, at least when using opaque access tokens: how else can the RS determine where it has to send the token introspection request. The RS model identifies the resource owner through the access token (see above). Then, the RS model creates a fresh nonce, which represents the protected resource of the resource owner. That nonce is then returned the client.

APPENDIX B
ADDITIONAL ATTACKS AND INCONSISTENCIES IN THE FAPI 2.0 ATTACKER MODEL

A. Authorization Request Leak Attack

In this attack, the authorization request (Step 8 in Figure 1) of an otherwise honest flow between an honest user u, an honest client $\mathtt{c}_{\mathsf{hon}}$, and honest AS $\mathtt{AS}_{\mathsf{hon}}$ is blocked by ($\mathcal{A}1$), but leaks to the attacker ($\mathcal{A}2$). The attacker now visits $\mathtt{AS}_{\mathsf{hon}}$'s authorization endpoint, posing as u with the leaked authorization request, and logs in using its own (attacker) identity. $\mathtt{AS}_{\mathsf{hon}}$ then answers with an authorization response, containing an authorization code $code$, which $\mathtt{AS}_{\mathsf{hon}}$ associates with the attacker identity and the PAR of the initial, honest flow. From the authorization response, the attacker assembles a link pointing to $\mathtt{c}_{\mathsf{hon}}$'s redirection endpoint with $code$ and the `iss` parameter as received from $\mathtt{AS}_{\mathsf{hon}}$, and sends this link to u. Once u follows this link, $\mathtt{c}_{\mathsf{hon}}$ exchanges the $code$ for tokens, logs u in under the attacker's identity and accesses resources belonging to the attacker in a session with u, hence violating both session integrity goals.

Fix. Upon our notification, the FAPI WG acknowledged the attack [3]. After some discussion, there was consensus that there is no currently deployed technology which can prevent such attacks. However, the FAPI WG wanted to keep $\mathcal{A}2$ in their attacker model. Therefore, the FAPI WG decided to document the attack, as well as (optional) mitigations, which do not prevent, but harden against the attack in practice, in the specifications [32, Section 5.6.6]. For our analysis, we formulated the session integrity properties such that they only apply to flows in which the authorization request does not leak.

B. Variants of Client Impersonation Attack

The attack described in Section III-B has two additional variants which we describe here.

Recall that in the attack variant described in Section III-B, the user authenticated at an honest, trusted AS and authorized not only an honest and trusted client, but also exactly the client which the user expected to authorize. However, if one assumes that the user does not thoroughly check which client they authorize, this attack even works if they initially contact a different client. In the first variant, we assumed that the user starts a flow with $\mathtt{c}_{\mathsf{hon}}$, i.e., an honest client of the user's choice, for which the attacker needs leaked client authentication JWTs (we describe how the attacker can obtain such JWTs in Section III-B). This variant, at the cost of the additional assumption that the user does not pay close attention to the client they are authorizing, makes the attack much more viable in practice: the attack can now select an arbitrary (but honest) client for which it can obtain client authentication JWTs. We note that this additional assumption about the user's behaviour is not a particularly strong one: even if the user pays attention, the AS also has to gather and show enough (reliable) information about the client, which can be quite difficult in practice, especially in dynamic environments, in which information AS has on clients are provided by the clients themselves in some automatic registration process.

In yet another variant of the attack, the attacker once again acts as $\mathtt{c}_{\mathsf{hon}}$ towards $\mathtt{AS}_{\mathsf{hon}}$, but sends a PAR without waiting for the user to initiate a flow. The attacker then constructs a link similar to Step 9 and uses social engineering to make the user click on that link. The remainder of the attack is the same. We note that this attack variant assumes that the attacker can convince the user to not only click a link and authenticate at $\mathtt{AS}_{\mathsf{hon}}$, but also to authorize $\mathtt{c}_{\mathsf{hon}}$ even though the user did not start a flow at all (however, $\mathtt{AS}_{\mathsf{hon}}$ and $\mathtt{c}_{\mathsf{hon}}$ may be entities the user knows and trusts).

C. Inconsistencies in Attacker Model

In addition to the attacks described above, we discovered inconsistencies in the FAPI 2.0 Attacker Model, i.e., assumptions on the attacker which immediately violate one or more of the security goals.

$\mathcal{A}6$ **Violates Authorization Goal.** As resource responses contain users' resources, this attacker assumption immediately violates the authorization security goal. When we pointed this out to the FAPI WG, it was decided to drop this attacker assumption [13].

$\mathcal{A}7$ **Violates Session Integrity for Authorization Goal.** If the attacker can tamper with resource responses, the attacker can force users to use attacker resources by replacing (honest) resources in resource responses with attacker resources. This obviously violates the session integrity for authorization goal. As above, the FAPI WG decided to drop the attacker assumption once we pointed out this inconsistency [13].

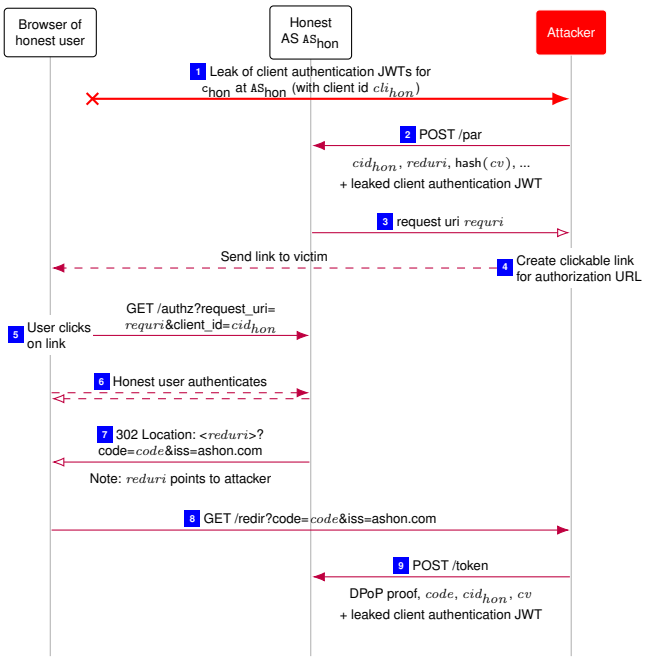

Figure 5. Variant of client impersonation attack

APPENDIX C
MODELING OF REMAINING ATTACKER ASSUMPTIONS

In Section IV-B2, we describe how we model the final assumptions on the attacker (i.e., as of the latest FAPI 2.0 Attacker Model). As mentioned there, our initial model contained all attacker assumptions as of July 1st (listed in Section II-C). We now detail how we incorporated the remaining assumptions (i.e., those which have been removed since July 1st) into our initial model and describe why we removed them from the final model.

$\mathcal{A}3$ (authorization responses leak) was initially modeled by leakage of the authorization response at the AS, which sent the response in plain to a non-deterministically chosen IP address (in addition to sending the response to the client). As described in Section III-D, we had to remove this leak according to the changed attacker model.

$\mathcal{A}4$ (attacker can trick client into using an attacker-controlled token endpoint) was initially modeled by the client non-deterministically choosing whether to use the correct token endpoint or a non-deterministically chosen endpoint. After the FAPI WG made it mandatory for clients to use server metadata, we removed this attacker capability (see Section III-B).

$\mathcal{A}6$ and $\mathcal{A}7$. As mentioned in Appendix B-C, the FAPI WG dropped these attacker assumptions after we reported inconsistencies in the attacker model resulting from these assumptions. Thus, we did not model these two assumptions.

APPENDIX D
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 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* [11] is modeled. We continue with the formal models of the FAPI 2.0 clients (Appendix D-J), the FAPI 2.0 ASs (Appendix D-K), and the FAPI 2.0 RSs (Appendix D-L).

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

B. Identities

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

Definition 5. 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 $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

C. 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. RSs use these to authenticate at ASs' 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 RS–AS pairs (with the function arguments in that order).

D. Passwords

Definition 6. 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).

E. Web Browsers

Web browser processes (i.e., processes $b \in \mathsf{B}$) are modeled as described in Appendix G. Before defining the initial states of Web browsers, we introduce the following set (for some process p):

$$\mathsf{Secrets}^{b,p} = \{s \mid b = \mathsf{ownerOfSecret}(s) \wedge (\exists i\colon s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{ID}^p)\}$$

Definition 7 (Initial Web Browser State for FAPI). The initial state of a Web browser process $b \in \mathsf{B}$ follows the description in Definition 68, with the following additional constraints:

- $s_0^b.\mathtt{ids} \equiv \langle\{i \mid b = \mathsf{ownerOfID}(i)\}\rangle$
- $s_0^b.\mathtt{secrets}$ contains an entry $\langle\langle d, \mathsf{S}\rangle, \langle\mathsf{Secrets}^{b,p}\rangle\rangle$ for each $p \in \mathsf{AS} \cup \mathsf{C} \cup \mathsf{RS}$ and every domain $d \in \mathsf{dom}(p)$ (and nothing else), i.e.,
$$s_0^b.\mathtt{secrets} \equiv \left\langle \left\{ \langle\langle d, \mathsf{S}\rangle, \langle\mathsf{Secrets}^{b,p}\rangle\rangle \;\middle|\; \exists p, d\colon p \in \mathsf{AS} \cup \mathsf{C} \cup \mathsf{RS} \wedge d \in \mathsf{dom}(p) \right\} \right\rangle$$
- $s_0^b.\mathtt{keyMapping} \equiv \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle \mid d \in \mathsf{Doms}\}\rangle$

F. Resources

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

1) identify the AS 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 RS retrieves the identity from the subject field. Otherwise, the identity is retrieved from the introspection response.

For identifying the AS, we first define the URL paths of resources managed by a RS, and then define a mapping from these paths to AS.

Definition 8. 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 RS.[5]

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

Definition 9. For each $rs \in \mathsf{RS}$, let $\mathsf{supportedAuthorizationServer}^{rs} \subseteq \mathsf{AS}$ be a finite set of ASs. These are the ASs supported by the RS.

Definition 10. For each $rs \in \mathsf{RS}$, let $\mathsf{authorizationServerOfResource}^{rs} : \mathsf{resourceURLPath}^{rs} \to \mathsf{supportedAuthorizationServer}^{rs}$ be a mapping that assigns an AS to each resource URL path suffix of resources managed by the RS.

If the access token is valid and the resource is managed by an AS supported by the RS, the RS model responds with a fresh nonce that it stores under the identity of the resource owner and the path under which it returns the resource. By using fresh nonces, the RS 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.

G. Client Registration

ASs and clients have to establish some relationship with each other before starting a FAPI 2.0 flow. Such a relationship can be established out of band, e.g., via manual configuration. While FAPI 2.0 also supports the use of dynamic client registration, our model assumes an out of band registration, captured by the following definitions.

Definition 11. 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 $\mathsf{ClientInfos}$ be the set of all client information dictionaries.

Definition 12. Let $\mathsf{clientInfo}\colon \mathsf{Doms} \times \mathsf{Doms} \rightharpoonup \mathsf{ClientInfos}$ be a (partial) mapping from AS and client domains to client information dictionaries, assigning a client information dictionary to some of the possible AS–client pairs with the following restrictions: 1) There are no two clients with the same $clientId$ at the same AS, 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 $\mathsf{signkey}$, 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 client to use the same key to sign JWTs and DPoP proofs for all ASs, it allows the client to use a different client type for each AS. mTLS keys are different for each of the client's domains.

Definition 13. 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 14. Let $\mathsf{clientInfoAS}\colon \mathsf{AS} \rightharpoonup \mathcal{T}_{\mathcal{N}}$ be a (partial) mapping from an AS to a dictionary. The keys of this dictionary are client IDs and the values *AS client information* dictionaries. We define $\mathsf{clientInfoAS}$ by $\mathsf{as} \mapsto \langle\{\langle cli.\texttt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid \exists\ d_{client} \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(\mathsf{as})\colon \mathsf{clientInfo}(d_{as}, d_{client}) \equiv cli\}\rangle$ 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 15. 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. $\mathsf{clientInfoClient}$ 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.

H. Modeling mTLS

OAuth 2.0 Mutual TLS for Client Authentication and Certificate Bound Access Tokens (mTLS) [11] 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*[6], which the client can use for client authentication at the pushed authorization request and token endpoints (in Step 5 and Step 14 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.

[6]As noted in Section 7.2 of [11], this extension supports all TLS versions with certificate-based client authentication.

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 18 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.[7]

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.

We model mTLS similarly to [38]. An overview of the mTLS model is shown in Figure 6. 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 AS. The AS 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 [68, Section 7.4.8], which also includes information about the server certificate, which means that the signature cannot be reused for another server. Instead of signing a sequence that contains information about the receiver, in our model, the client checks the sender of the nonce, and only sends the decrypted nonce to the creator of the nonce. In other words, a nonce decrypted by an honest server that gets decrypted by the honest client is never sent to the attacker.

As explained above, the client uses the same certificate it used for the token request when sending the access token to the RS. While the RS has to check the possession of corresponding private keys, the validity of the certificate was already checked at the AS and can be ignored by the RS. Therefore, in our model of FAPI 2.0, the client does not send its client id to the RS, but its public key, and the RS encrypts the message with this public key.

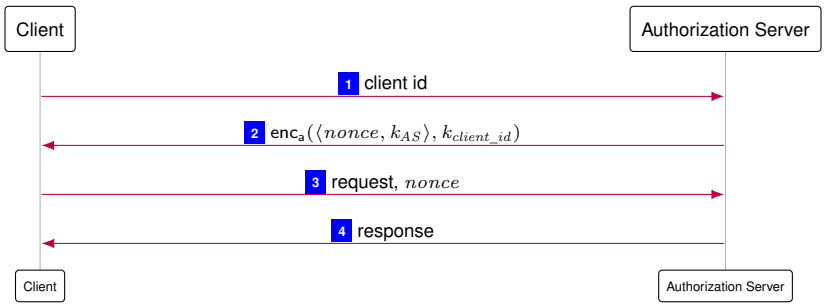

Figure 6. Overview of mTLS model

All messages are sent by the generic HTTPS server model (Appendix G-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 7 shows the complete messages, i.e., with the encryption used for transmitting the messages.

I. Additional HTTP Headers

In order to model DPoP, we extend the list of headers of Definition 42 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).

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

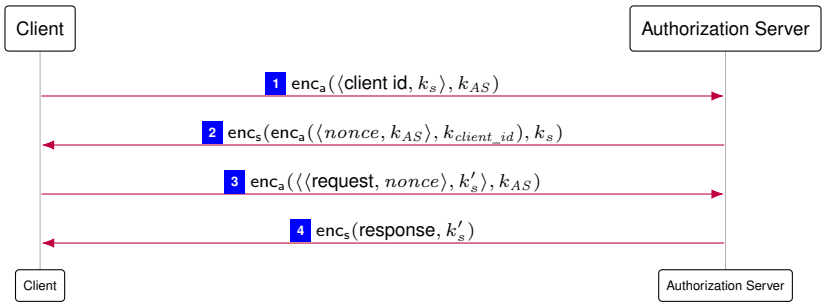

Figure 7. Detailed view on mTLS model

J. 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 16. 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 71), $sessions \in [\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $oauthConfigCache \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $jwksCache \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $asAccounts \in [\mathcal{T}_\mathcal{N} \times \mathcal{T}_\mathcal{N}]$, $mtlsCache \in \mathcal{T}_\mathcal{N}$, $jwk \in K_{\text{sign}}$, $resourceASMapping \in [\mathsf{Doms} \times [\mathbb{S} \times \mathsf{Doms}]]$, and $dpopNonces \in [\mathsf{Doms} \times \mathcal{T}_\mathcal{N}]$.

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

- $s_0^c.\texttt{DNSAddress} \in \mathsf{IPs}$,
- $s_0^c.\texttt{pendingDNS} \equiv \langle\rangle$,
- $s_0^c.\texttt{pendingRequests} \equiv \langle\rangle$,
- $s_0^c.\texttt{corrupt} \equiv \bot$,
- $s_0^c.\texttt{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^c.\texttt{tlskeys} \equiv tlskeys^c$ (see Appendix D-C),
- $s_0^c.\texttt{sessions} \equiv \langle\rangle$,
- $s_0^c.\texttt{oauthConfigCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{jwksCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{asAccounts} \equiv \mathsf{clientInfoClient}(c)$ (see Definition 15),
- $s_0^c.\texttt{mtlsCache} \equiv \langle\rangle$,
- $s_0^c.\texttt{jwk} \equiv \mathsf{signkey}(c)$ (see Appendix D-C),
- $s_0^c.\texttt{resourceASMapping}[domRS][resourceID] \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID))$, $\forall\, rs \in \mathsf{RS}$ and $\forall\, domRS \in \mathsf{dom}(rs)$ and $\forall\, resourceID \in \mathsf{resourceURLPath}^{rs}$ (a domain of the AS 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 the model of generic HTTPS servers (see Appendix G-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 the communication model (see Definition 27).

The script that is used by the client on its index page is specified in Algorithm 8. This script uses the $\mathsf{GETURL}(tree, docnonce)$ function to to extract the current URL of a document. 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.`path` $\equiv$ `/` **then** → Serve index page (start flow).
    let $m' := \mathsf{enc_s}(\langle$`HTTPResp`$, m.$`nonce`$, 200, headers, \langle$`script_client_index`$, \langle\rangle\rangle, k)$ → Reply with *script_client_index*.
    stop $\langle\langle f, a, m'\rangle\rangle, s'$
  else if m.`path` $\equiv$ `/startLogin` $\wedge$ m.`method` $\equiv$ `POST` **then** → Start a new FAPI 2.0 flow (probably from `script_client_index`)
    if m.`headers`[`Origin`] $\not\equiv \langle m.$`host`, `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 $selectedAS := m.$`body`
    let $sessionId := \nu_1$ → Session id is a freshly chosen nonce.
    let $s'.$`sessions`[$sessionId$] := [`startRequest`: [`message`: m, `key`: k, `receiver`: a, `sender`: f], `selected_AS`: $selectedAS$]
    call PREPARE_AND_SEND_PAR($sessionId$, a, s') → Start an authorization flow with the AS (see Algorithm 6)
  else if m.`path` $\equiv$ `/redirect_ep` **then** → User is being redirected after authentication to the AS.
    let $sessionId := m.$`headers`[`Cookie`][$\langle$`__Host`, `sessionId`$\rangle$]
    if $sessionId \notin s'.$`sessions` **then**
      stop
    let $session := s'.$`sessions`[$sessionId$] → Retrieve session data.
    let $selectedAS := session$[`selected_AS`]
    if $m.$`parameters`[`iss`] $\not\equiv selectedAS$ **then** → Check issuer parameter ([75]).
      stop
    → Store browser's request for use in CHECK_ID_TOKEN (Algorithm 5) and PROCESS_HTTPS_RESPONSE (Algorithm 2)
    let $s'.$`sessions`[$sessionId$][`redirectEpRequest`] := [`message`: m, `key`: k, `receiver`: a, `sender`: f]
    call SEND_TOKEN_REQUEST($sessionId$, $m.$`parameters`[`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 Appendix D-H)
    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` identifier according to [48, Sec. 3.3]
      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 ([41, Sec. 3.1]). However, following [57, Sec. 4] and [73, Sec. 5] 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`] := $[\langle \texttt{\_\_Host}, \texttt{sessionId} \rangle : \langle sessionId, \top, \top, \top \rangle]$
    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 [30] and Section IV-B2.
    if *leakAuthZReq* $\equiv \top$ **then**
      let *leak* := $\langle \texttt{LEAK}, authEndpoint \rangle$
      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 AS (see [70, 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 $\langle \langle request[\texttt{sender}], request[\texttt{receiver}], m' \rangle \rangle, s'$

→ Algorithm continues on next page.

```
    else if reference[responseTo] ≡ DPOP_NONCE then
        let dpopNonce := m.body[nonce]
        let rsDomain := request.host
        let s'.dpopNonces[rsDomain] := s'.dpopNonces[rsDomain] +^⟨⟩ dpopNonce
        stop ⟨⟩, s'
    stop
```

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

```
function SEND_TOKEN_REQUEST(sessionId, code, a, s')
    let session := s'.sessions[sessionId]
    let pkceVerifier := session[code_verifier]
    let selectedAS := session[selected_AS]
    let headers := []
    let body := [grant_type: authorization_code, code: code, redirect_uri: session[redirect_uri]]
    let body[code_verifier] := pkceVerifier    → add PKCE Code Verifier ([72], Section 4.5)
    let clientId := s'.asAccounts[selectedAS][client_id]
    let clientType := s'.asAccounts[selectedAS][client_type]
    let oauthConfig := s'.oauthConfigCache[selectedAS]
    let tokenEndpoint := oauthConfig[token_ep]
    → Client Authentication:
    if clientType ∈ {mTLS_mTLS, mTLS_DPoP} then    → mTLS client authentication
        let body[client_id] := clientId    → [11] mandates client_id when using mTLS authentication
        let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
        let authData := [TLS_AuthN: mtlsNonce]
        let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
    else if clientType ∈ {pkjwt_mTLS, pkjwt_DPoP} then    → private_key_jwt client authentication
        let jwt := [iss: clientId, sub: clientId, aud: selectedAS]
        let jws := sig(jwt, s'.jwk)
        let authData := [client_assertion: jws]
    else
        stop    → Invalid client type
    → Sender Constraining:
    if clientType ≡ mTLS_mTLS then    → mTLS sender constraining (same nonce as for mTLS authN)
        let mtlsNonce := authData[TLS_AuthN]
        let body[TLS_binding] := mtlsNonce
    else if clientType ≡ pkjwt_mTLS then    → mTLS sender constraining (fresh mTLS nonce)
        let mtlsNonce such that ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩ ∈ s'.mtlsCache if possible; otherwise stop
        let s'.mtlsCache := s'.mtlsCache −^⟨⟩ ⟨tokenEndpoint.host, clientId, ⟨⟩, mtlsNonce⟩
        let body[TLS_binding] := mtlsNonce
    else    → Sender constraning using DPoP
        let privKey := s'.jwk    → get private key
        let htu := tokenEndpoint
        let htu[parameters] := ⟨⟩    → [39, Sec. 4.2]: without query
        let htu[fragment] := ⊥    → [39, Sec. 4.2]: without fragment
        let dpopJwt := [headers: [jwk: pub(privKey)]]
        let dpopJwt[payload] := [htm: POST, htu: htu]
        let dpopProof := sig(dpopJwt, privKey)
        let headers[DPoP] := dpopProof    → add DPoP header; the dpopJwt can be extracted with the extractmsg() function
    let body := body +^⟨⟩ authData
    let message := ⟨HTTPReq, ν2, POST, tokenEndpoint.host, tokenEndpoint.path, tokenEndpoint.parameters, headers, body⟩
    call HTTPS_SIMPLE_SEND([responseTo: TOKEN, session: sessionId], message, a, s')
```

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

function USE_ACCESS_TOKEN($sessionId$, $token$, $tokenEPDomain$, a, s')
  let $session := s'.\texttt{sessions}[sessionId]$
  let $selectedAS := session[\texttt{selected\_AS}]$
  let $rsDomain \leftarrow \mathsf{Doms}$ → This domain may or may not belong to a "real" RS. 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 RS.
  let $resourceID \leftarrow \mathbb{S}$ **such that** $resourceID \notin \{/\texttt{MTLS-prepare}, /\texttt{DPoP-nonce}\}$
  let $url := \langle \texttt{URL}, \texttt{S}, rsDomain, resourceID, \langle\rangle, \bot \rangle$
  if $s'.\texttt{resourceASMapping}[rsDomain][resourceID] \not\equiv tokenEPDomain$ **then**
    stop → The AS from which the client received the AT is not managing the resource
  → The access token is sender-constrained, so the client must add a corresponding key proof.
  let $clientType := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_type}]$
  let $clientId := s'.\texttt{asAccounts}[selectedAS][\texttt{client\_id}]$
  let $body := []$
  if $clientType \in \{\texttt{mTLS\_mTLS}, \texttt{pkjwt\_mTLS}\}$ **then** → mTLS sender constraining
    let $mtlsNonce$ **such that** $\langle rsDomain, \langle\rangle, pubKey, mtlsNonce\rangle \in s'.\texttt{mtlsCache}$ **if possible; otherwise stop**
    let $body[\texttt{TLS\_binding}] := mtlsNonce$ → This nonce is not necessarily associated with the same of the client's keys as the access token. In such a case, the RS will reject this request and the client has to try again.
    let $headers := [\texttt{Authorization}: [\texttt{Bearer}: token]]$ → FAPI 2.0 mandates to send access token in header
    let $s'.\texttt{mtlsCache} := s'.\texttt{mtlsCache} -^{\langle\rangle} \langle rsDomain, \langle\rangle, pubKey, mtlsNonce\rangle$
  else if $clientType \in \{\texttt{mTLS\_DPoP}, \texttt{pkjwt\_DPoP}\}$ **then** → DPoP sender constraining
    let $privKey := s'.\texttt{jwk}$ → get private key
    let $dpopNonce$ **such that** $dpopNonce \in s'.\texttt{dpopNonces}[rsDomain]$ **if possible; otherwise stop**
    let $s'.\texttt{dpopNonces}[rsDomain] := s'.\texttt{dpopNonces}[rsDomain] -^{\langle\rangle} dpopNonce$
    let $htu := url$
    let $htu[\texttt{parameters}] := \langle\rangle$ → [39, Sec. 4.2]: without query
    let $htu[\texttt{fragment}] := \bot$ → [39, Sec. 4.2]: without fragment
    let $dpopJwt := [\texttt{headers}: [\texttt{jwk}: \mathsf{pub}(privKey)]]$
    let $dpopJwt[\texttt{payload}] := [\texttt{htm}: \texttt{POST}, \texttt{htu}: htu, \texttt{ath}: \mathsf{hash}(token), \texttt{nonce}: dpopNonce]$
    let $dpopProof := \mathsf{sig}(dpopJwt, privKey)$
    let $headers := [\texttt{Authorization}: [\texttt{DPoP}: token]]$ → See [39, Sec. 7.1]
    let $headers[\texttt{DPoP}] := dpopProof$ → add DPoP header; the $dpopJwt$ can be extracted with the extractmsg() function
  let $message := \langle \texttt{HTTPReq}, \nu_3, \texttt{POST}, url.\texttt{domain}, url.\texttt{path}, \langle\rangle, headers, body\rangle$
  call HTTPS_SIMPLE_SEND($[\texttt{responseTo}: \texttt{RESOURCE\_USAGE}, \texttt{session}: sessionId]$, $message$, a, s')

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

function CHECK_ID_TOKEN($sessionId$, $idToken$, s') → **Check ID Token validity and create service session.**
  let $session := s'.\texttt{sessions}[sessionId]$ → Retrieve session data.
  let $selectedAS := session[\texttt{selected\_AS}]$
  let $oauthConfig := s'.\texttt{oauthConfigCache}[selectedAS]$ → Retrieve configuration for user-selected AS.
  let $clientInfo := s'.\texttt{asAccounts}[selectedAS]$ → Retrieve client info used at that AS.
  let $jwks := s'.\texttt{jwksCache}[selectedAS]$ → Retrieve signature verification key for AS.
  let $data := \mathsf{extractmsg}(idToken)$ → Extract contents of signed ID Token.
  → The following ID token checks are mandated by [70, 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 AS (which it is here). Hence, we do not check the token's signature (see also Line 47 of Algorithm 2).
  if $data[\texttt{iss}] \not\equiv selectedAS$ **then**
    stop → Check the issuer; note that previous checks ensure $oauthConfig[\texttt{issuer}] \equiv selectedAS$
  if $data[\texttt{aud}] \not\equiv clientInfo[\texttt{client\_id}]$ **then**
    stop → Check the audience against own client id.
  if $nonce \in session \wedge data[\texttt{nonce}] \not\equiv session[\texttt{nonce}]$ **then**
    stop → If a nonce was used, check its value.
  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.
  let $s'.\texttt{sessions}[sessionId][\texttt{serviceSessionId}] := \nu_4$ → Choose a new service session id.
  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).
  let $headers[\texttt{Set-Cookie}] := [\texttt{serviceSessionId}: \langle \nu_4, \top, \top, \top\rangle]$ → Create a cookie containing the service session id, effectively logging the user identified by $data[\texttt{sub}]$ in at this client.
  let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, request[\texttt{message}].\texttt{nonce}, 200, headers, \texttt{ok}\rangle, request[\texttt{key}])$
  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 $selectedAS := session[\texttt{selected\_AS}]$ → AS selected by the user at the beginning of the flow.
  → Check whether the client needs to fetch AS metadata first and do so if required.
  if $selectedAS \notin s'.\texttt{oauthConfigCache}$ **then**
    let $path \leftarrow \{/\texttt{.well\_known/openid-configuration}, /\texttt{.well\_known/oauth-authorization-server}\}$
    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 AS'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 $jwt := [\texttt{iss}: clientId, \texttt{sub}: clientId, \texttt{aud}: selectedAS]$
    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 $parData := [\texttt{response\_type}: \texttt{code}, \texttt{code\_challenge\_method}: \texttt{S256}, \texttt{client\_id}: clientId,$
    $\hookrightarrow\ \texttt{redirect\_uri}: redirectUri, \texttt{code\_challenge}: pkceChallenge]$
  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" RS.
      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 RS. The RS model offers a special endpoint to retrieve nonces.
      let $resourceServer \leftarrow$ Doms → Note: This may or may not be a "real" RS.
      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 $url' := \langle$`URL`, `S`, $url.$`host`, /`startLogin`, $\langle\rangle$, $\perp\rangle$ → Assemble request URL.
  let $command := \langle$`FORM`, url', `POST`, as, $\perp\rangle$ → 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 $url := \langle$`URL`, $protocol$, $host$, $path$, $parameters$, $fragment\rangle$ → Assemble request URL.
  let $command := \langle$`HREF`, url, $\perp$, $\perp\rangle$ → 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).

K. 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 17. A *state* $s \in Z^{\mathsf{as}}$ *of an AS* as is a term of the form $\langle DNSAddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, jwk, registrationRequests, clients, records, authorizationRequests, mtlsRequests, rsCredentials\rangle$ 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 71), $tlskeys \in [\mathsf{Doms} \times K_{\text{TLS}}]$, $jwk \in K_{\text{sign}}$, $registrationRequests \in \mathcal{T}_{\mathcal{N}}$, $clients \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $records \in \mathcal{T}_{\mathcal{N}}$, $authorizationRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, $mtlsRequests \in \left[\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}\right]$, and $rsCredentials \in \mathcal{T}_{\mathcal{N}}$.

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

- $s_0^{\mathsf{as}}.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^{\mathsf{as}}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{\mathsf{as}}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{\mathsf{as}}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{\mathsf{as}}.\mathtt{keyMapping}$ being the same as the keymapping for browsers,
- $s_0^{\mathsf{as}}.\mathtt{tlskeys} \equiv tlskeys^{\mathsf{as}}$,
- $s_0^{\mathsf{as}}.\mathtt{jwk} \equiv \mathsf{signkey}(\mathsf{as})$ (see Appendix D-C),
- $s_0^{\mathsf{as}}.\mathtt{registrationRequests} \equiv \langle\rangle$,
- $s_0^{\mathsf{as}}.\mathtt{clients} \equiv \mathsf{clientInfoAS}(\mathsf{as})$ (see Definition 14),
- $s_0^{\mathsf{as}}.\mathtt{records} \equiv \langle\rangle$,
- $s_0^{\mathsf{as}}.\mathtt{authorizationRequests} \equiv \langle\rangle$,
- $s_0^{\mathsf{as}}.\mathtt{mtlsRequests} \equiv \langle\rangle$, and
- $s_0^{\mathsf{as}}.\mathtt{rsCredentials} \equiv rsCreds$ 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 the model of generic HTTPS servers (see Appendix G-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 $m.\texttt{path} \equiv /.\texttt{well-known/openid-configuration} \vee m.\texttt{path} \equiv /.\texttt{well-known/oauth-authorization-server}$ **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** $s'.\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}] \equiv auth2Reference$ **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

→ Algorithm continues on next page.

```
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
```

→ Algorithm continues on next page.

```
        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 := ν2    → 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    → RS 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 Appendix D-H
        let clientId := m.body[client_id]
        let mtlsNonce := ν3
        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 $clientId := m.\texttt{body}[\texttt{client\_id}]$ → [11] 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 $\langle s, cookies, localStorage, sessionStorage, command\rangle$

L. 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 18. A *state* $s \in Z^{rs}$ *of a resource server* rs is a term of the form $\langle DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, mtlsRequests, pendingResponses, resourceNonces, ids, asInfo, resourceASMapping, dpopNonces\rangle$ with $DNSaddress \in \mathsf{IPs}$, $pendingDNS \in \big[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\big]$, $pendingRequests \in \big[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\big]$, $corrupt \in \mathcal{T}_{\mathcal{N}}$, $keyMapping \in \big[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\big]$, $tlskeys \in [\mathsf{Doms} \times K_{\mathrm{TLS}}]$ (all former components as in Definition 71), $mtlsRequests \in \mathcal{T}_{\mathcal{N}}$, $pendingResponses \in \mathcal{T}_{\mathcal{N}}$, $resourceNonces \in \big[\mathsf{ID} \times \mathcal{T}_{\mathcal{N}}\big]$, $ids \subset \mathsf{ID}$, $asInfo \in \big[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\big]$, $resourceASMapping \in \big[\mathsf{resourceURLPath}^{rs} \times \mathcal{T}_{\mathcal{N}}\big]$, and $dpopNonces \in \mathcal{T}_{\mathcal{N}}$.

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

- $s_0^{rs}.\mathtt{DNSaddress} \in \mathsf{IPs}$,
- $s_0^{rs}.\mathtt{pendingDNS} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{pendingRequests} \equiv \langle\rangle$,
- $s_0^{rs}.\mathtt{corrupt} \equiv \bot$,
- $s_0^{rs}.\mathtt{keyMapping}$ 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 RS stores the resource nonces for each identity and resource id pair, initialized as $s_0^{rs}.\mathtt{resourceNonces}[id][resourceID] := \langle\rangle, \forall id \in^{\langle\rangle} s_0^{rs}.\mathtt{ids}, \forall resourceID \in \mathbb{S}$
- $s_0^{rs}.\mathtt{ids} \subset^{\langle\rangle} \langle\mathsf{ID}\rangle$ such that $\forall id \in s_0^{rs}.\mathtt{ids} : \mathsf{governor}(id) \in \mathsf{supportedAuthorizationServer}^{rs}$, i.e., the RS manages only resources of identities that are governed by one of the AS supported by the RS,
- and for each domain of a supported AS $dom_{as} \in \mathsf{supportedAuthorizationSeverDoms}^{rs}$, 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 AS)
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{as\_key}] \equiv \mathsf{signkey}(\mathsf{dom}^{-1}(dom_{as}))$ being the verification key for the AS
  - $s_0^{rs}.\mathtt{asInfo}[dom_{as}][\mathtt{rs\_credentials}]$ 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 RS for authenticating at the AS.
- $s_0^{rs}.\mathtt{resourceASMapping}[resourceID] \in \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID))$, $\forall resourceID \in \mathsf{resourceURLPath}^{rs}$ (a domain of the AS 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 G-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

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:     if m.path ≡ /MTLS-prepare then
3:         let mtlsNonce := ν_1
4:         let clientKey := m.body[pub_key]    → Certificate is not required to be checked [11, Section 4.2]
5:         let s'.mtlsRequests := s'.mtlsRequests +^⟨⟩ ⟨mtlsNonce, clientKey⟩
6:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, enc_a(⟨mtlsNonce, s'.keyMapping[m.host]⟩, clientKey)⟩, k)
7:         stop ⟨⟨f, a, m'⟩⟩, s'
8:     else if m.path ≡ /DPoP-nonce then
9:         let freshDpopNonce := ν_dpop
10:        let s'.dpopNonces := s'.dpopNonces +^⟨⟩ freshDpopNonce
11:        let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [nonce: freshDpopNonce]⟩, k)
12:        stop ⟨⟨f, a, m'⟩⟩, s'
13:    else
14:        let resourceID := m.path
15:        let responsibleAS := s'.resourceASMapping[resourceID]
16:        if responsibleAS ≡ ⟨⟩ then
17:            stop    → Resource is not managed by any of the supported ASs
18:        let asInfo := s'.asInfo[responsibleAS]
19:        if Authorization ∈ m.headers then
20:            let authnScheme := m.headers[Authorization].1
21:            let accessToken := m.headers[Authorization].2
22:            if authnScheme ≡ Bearer then    → mTLS sender constraining
23:                let mtlsNonce := m.body[TLS_binding]
24:                let mtlsInfo such that mtlsInfo ∈^⟨⟩ s'.mtlsRequests ∧ mtlsInfo.1 ≡ mtlsNonce if possible; otherwise stop
25:                let s'.mtlsRequests := s'.mtlsRequests −^⟨⟩ mtlsInfo
26:                let mtlsKey := mtlsInfo.2
27:                let cnfValue := [x5t#S256: hash(mTlsKey)]
28:            else if authnScheme ≡ DPoP then    → DPoP sender constraining
29:                let dpopProof := m.headers[DPoP]
30:                let dpopJwt := extractmsg(dpopProof)
31:                let verificationKey := dpopJwt[headers][jwk]
32:                if checksig(dpopProof, verificationKey) ≢ ⊤ ∨ verificationKey ≡ ⟨⟩ then
33:                    stop    → Invalid DPoP signature (or empty jwk header)
34:                let dpopClaims := dpopJwt[payload]
35:                let reqUri := ⟨URL, S, m.host, m.path, ⟨⟩, ⊥⟩
36:                if dpopClaims[htm] ≢ m.method ∨ dpopClaims[htu] ≢ reqUri then
37:                    stop    → DPoP claims do not match corresponding message
38:                if dpopClaims[nonce] ∉ s'.dpopNonces then
39:                    stop    → Invalid DPoP nonce
40:                if dpopClaims[ath] ≢ hash(accessToken) then
41:                    stop    → Invalid access token hash
42:                let s'.dpopNonces := s'.dpopNonces −^⟨⟩ dpopClaims[nonce]
43:                let cnfValue := [jkt: hash(verificationKey)]
44:            else
45:                stop    → Wrong Authorization header value
46:            let resource := ν_4    → Generate a fresh resource nonce
47:            let accessTokenContent such that accessTokenContent ≡ extractmsg(accessToken)
                   ↪ if possible; otherwise let accessTokenContent := ⋄
48:            if accessTokenContent ≡ ⋄ then    → Not a structured AT, do Token Introspection
   → Store values for the pending request (needed when the RS gets the introspection response)
49:                let requestId := ν_2
50:                let s'.pendingResponses[requestId] := [expectedCNF: cnfValue, requestingClient: f,
                       ↪ resourceID: resourceID, originalRequest: m, originalRequestKey: k, resource: resource]
51:                let url := asInfo[as_introspect_ep]
52:                let rsCred ← asInfo[rs_credentials]    → Secret for authenticating at the AS (see also [69, Sec. 2.1])
53:                let headers := [Authorization: ⟨Basic, rsCred⟩]
54:                let body := [token: accessToken]
55:                let message := ⟨HTTPReq, ν_3, POST, url.domain, url.path, url.parameters, headers, body⟩
56:                call HTTPS_SIMPLE_SEND([responseTo: TOKENINTROSPECTION, requestId: requestId], message, a, s')
57:            else    → Check structured AT
58:                if cnfValue.1 ≢ accessTokenContent[cnf].1 ∨ cnfValue.2 ≢ accessTokenContent[cnf].2 then
59:                    stop    → AT is bound to a different key
60:                if checksig(accessToken, asInfo[as_key]) ≢ ⊤ then
61:                    stop    → Verification of AT signature failed
```

→ Algorithm continues on next page.

```
            let id := accessTokenContent[sub]
            if id ∉^⟨⟩ s'.ids then
                stop      → RS does not manage resources of this RO
            → Token binding successfully checked, the RS gives access to a resource of the identity
            let s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] +^⟨⟩ resource
            let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, [resource:resource]⟩, k)
        → Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
            let leakingMessage := ⟨HTTPReq, ν_RRleak, POST, m.domain, m.path, m.parameters, m.headers, []⟩
            let leakAddress ← IPs
            stop ⟨⟨f, a, m'⟩, ⟨leakAddress, a, ⟨LEAK, leakingMessage⟩⟩⟩, s'
        else
            stop      → Expected AT in Authorization header as mandated by FAPI 2.0
```

Algorithm 13 Relation of a Resource Server R^{rs} – Processing HTTPS Responses

```
1:  function PROCESS_HTTPS_RESPONSE(m, reference, request, key, a, f, s')
2:      if reference[responseTo] ≡ TOKENINTROSPECTION then
3:          let pendingRequestInfo := s'.pendingResponses[reference[requestId]]
4:          let clientAddress := pendingRequestInfo[requestingClient]
5:          let expectedCNF := pendingRequestInfo[expectedCNF]
6:          let origReq := pendingRequestInfo[originalRequest]
7:          let originalRequestKey := pendingRequestInfo[originalRequestKey]
8:          let resourceID := pendingRequestInfo[resourceID]
9:          let resource := pendingRequestInfo[resource]
10:         if m.body[active] ≢ ⊤ then
11:             stop    → Access token was invalid
12:         let responseCNF := m.body[cnf]
13:         if responseCNF.1 ≢ expectedCNF.1 ∨ responseCNF.2 ≢ expectedCNF.2 then
14:             stop    → Access token was bound to a different key
15:         let id := m.body[sub]
16:         if id ∉^⟨⟩ s'.ids then
17:             stop      → RS does not manage resources of this RO
        → Token binding successfully checked, the RS gives access to a resource of the identity
18:         let s'.resourceNonces[id][resourceID] := s'.resourceNonces[id][resourceID] +^⟨⟩ resource
19:         let m' := enc_s(⟨HTTPResp, origReq.nonce, 200, ⟨⟩, [resource:resource]⟩, originalRequestKey)
            → Leak resource request. Note that we only leak the application-layer message, and in particular, not the mTLS nonce.
20:         let leakingMessage := ⟨HTTPReq, ν_RRleak, POST, origReq.domain, origReq.path, origReq.parameters, origReq.headers, []⟩
21:         let leakAddress ← IPs
22:         stop ⟨⟨f, a, m'⟩, ⟨leakAddress, a, ⟨LEAK, leakingMessage⟩⟩⟩, s'
```

APPENDIX E
FORMAL SESSION INTEGRITY PROPERTIES

In Section IV-E, we present our authorization and authentication properties. Here, we give the formal definition of the session integrity properties.

A. Session Integrity

On a high-level view, 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.

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.

More precisely, 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 run of a FAPI 2.0 web system.

Definition 19 (User is logged in). For a run ρ of a FAPI 2.0 web system with 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 $\langle \texttt{Set-Cookie}, [\texttt{serviceSessionId}\colon \langle ssid, \top, \top, \top\rangle]\rangle$ for some nonce $ssid$ such that $S(c).\texttt{sessions}[lsid][\texttt{serviceSessionId}] \equiv ssid$ and $S(c).\texttt{sessions}[lsid][\texttt{loggedInAs}] \equiv \langle d, id\rangle$ with $d \in \mathsf{dom}(as)$. We then write $\mathsf{loggedIn}^{Q}_{\rho}(b, c, id, as, lsid)$.

Definition 20 (User started a login flow). For a run ρ of a FAPI 2.0 web system with 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 $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionId}\rangle\colon \langle lsid, \top, \top, \top\rangle]\rangle$. We then write $\mathsf{started}^{Q}_{\rho}(b, c, lsid)$.

Definition 21 (User authenticated at an AS). For a run ρ of a FAPI 2.0 web system with 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 $scriptstate[\texttt{auth2\_reference}] \equiv auth2Reference$, and
- there is a nonce $requestUri$ such that $S(as).\texttt{authorizationRequests}[requestUri][\texttt{auth2\_reference}] \equiv auth2Reference$, and
- $S(c).\texttt{sessions}[lsid][\texttt{request\_uri}] \equiv requestUri$.

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

Definition 22 (Resource Access). For a run ρ of a FAPI 2.0 web system with 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 23 (Client Leaked Authorization Request). Let $\mathcal{FAPI}$ be a FAPI 2.0 web system with 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 $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$, 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 24 (Session Integrity for Authentication). Let $\mathcal{FAPI}$ be a FAPI 2.0 web system with 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 $\mathsf{loggedIn}_\rho^Q(b, c, id, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 23), we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, id, as, lsid)$.

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 25 (Session Integrity for Authorization). Let $\mathcal{FAPI}$ be a FAPI 2.0 web system with 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 $\mathsf{accessesResource}_\rho^{\mathsf{Q}}(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 23), , then (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}_\rho^{Q'}(b, c, lsid)$, and (2) if as is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q''}(b, c, u, as, lsid)$.

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

APPENDIX F
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

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

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 16, Definition 17, Definition 18). ■

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:

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

PROOF. $\mathsf{signkey}(c) \equiv s_0^c.\mathtt{jwk}$ immediately follows from Definition 16. $\mathsf{signkey}(c) \equiv S(c).\mathtt{jwk}$ follows from Definition 16 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.

By definition of $\mathsf{signkey}$ in Appendix D-C and the initial states of authorization servers (Definition 17), clients (Definition 16), browsers (Definition 7), and resource servers (Definition 18), we have that no other process initially knows $\mathsf{signkey}(c)$.

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

Lemma 3 (Client's TLS Keys 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, every domain $d_c \in \mathsf{dom}(c)$, and every process p with $p \neq c$, all of the following hold true:

1) $\mathsf{tlskey}(d_c) \notin d_\emptyset(S(p))$
2) $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} S^0(c).\mathtt{tlskeys}$
3) $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} S(c).\mathtt{tlskeys}$

PROOF. With Definition 16, $\langle d_c, \mathsf{tlskey}(c)\rangle \in^{\langle\rangle} S^0(c).\mathtt{tlskeys}$ is equivalent to $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} tlskeys^c$. This, in turn follows immediately from the definition of $tlskeys^c$ in Appendix D-C. Building on this, it is easy to check that the client never changes the contents of its `tlskeys` state subterm, i.e., we have $\langle d_c, \mathsf{tlskey}(d_c)\rangle \in^{\langle\rangle} S(c).\mathtt{tlskeys}$.

The only places in which an honest client accesses any value in its `tlskeys` state subterm are:

Line 3 of Algorithm 2 Here, the value is only used to decrypt a message (i.e., cannot leak).

Line 13 of Algorithm 7 Here, the client only uses the value to create a public key. As the equational theory does not allow extraction of private keys from public keys, it does not matter where that public key is stored or sent to.

Line 7 of Algorithm 31 Here, the value is once again only used to decrypt a message.

By definition of tlskey, $tlskeys^p$ in Appendix D-C and the initial states of authorization servers (Definition 17), clients (Definition 16), browsers (Definition 7), and resource servers (Definition 18), we have that no other process initially knows $\mathsf{tlskey}(d_c)$.

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

Lemma 4 (Generic Server - Correctness of Reference and Request). For any run ρ of a FAPI web system $\mathcal{FAPI}$ with a network attacker, every processing step $P = (S^P, E^P, N^P) \rightarrow (S^{P\prime}, E^{P\prime}, N^{P\prime})$ in ρ, every $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ being honest in S, it holds true that if p calls PROCESS_HTTPS_RESPONSE in P with $reference$ being the second and $request$ being the third input argument, then there exists a previous processing step in which p calls HTTPS_SIMPLE_SEND with $reference$ being the first and $request$ being the second input argument.

PROOF. Let $p \in \mathsf{C} \cup \mathsf{AS} \cup \mathsf{RS}$ be honest in S^P. p calls the PROCESS_HTTPS_RESPONSE function only in the generic HTTPS server algorithm in Line 26 of Algorithm 31. The values $reference$ and $request$ are taken from $S^P(p).\mathtt{pendingRequests}$ in Line 19 of Algorithm 31. Thus, p added these values to `pendingRequests` in a previous processing step $O = (S^O, E^O, N^O) \rightarrow (S^{O\prime}, E^{O\prime}, N^{O\prime})$ by executing Line 15 of Algorithm 31, as this is the only location where a client, authorization server, or resource server adds entries to `pendingRequests` and as `pendingRequests` is initially empty (see Definitions 16, 17, and 18). In O, the process p takes both values from $S^O(p).\mathtt{pendingDNS}$ in Line 13 and Line 14 of Algorithm 31. Initially `pendingDNS` is empty (as p is a client, an authorization server, or a resource server), and p adds values to `pendingDNS` only in Line 2 of Algorithm 26,

where the reference and request values are the input arguments of HTTPS_SIMPLE_SEND. Thus, in some processing step prior to O, p called HTTPS_SIMPLE_SEND with *reference* being the first and *request* being the second input argument.

■

Lemma 5 (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

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

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* being the second and *request* being the third input argument, and if $\mathit{reference}[\texttt{responseTo}] \equiv \texttt{TOKEN}$, then there is a previous configuration $(S^{L'}, E^{L'}, N^{L'})$ such that $\mathit{request}.\texttt{body}[\texttt{code}] \equiv S^{L'}(c).\texttt{sessions}[\mathit{reference}[\texttt{session}]][\texttt{redirectEpRequest}][\texttt{message}].\texttt{parameters}[\texttt{code}]$.

PROOF. As shown in Lemma 4, there exists a processing step $L = (S^L, E^L, N^L) \to (S^{L'}, E^{L'}, N^{L'})$ prior to P in which c called HTTPS_SIMPLE_SEND with the same reference and request values. The only line in which a client calls HTTPS_SIMPLE_SEND with $\mathit{reference}[\texttt{responseTo}] \equiv \texttt{TOKEN}$ is Line 41 of Algorithm 3 (SEND_TOKEN_REQUEST). The code included in the request is the input parameter of SEND_TOKEN_REQUEST (see Lines 6, 39, and 40 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 $S^{L'}(c).\texttt{sessions}[\mathit{sessionId}][\texttt{redirectionEpRequest}][\texttt{message}]$ in Line 20 of Algorithm 1, with $\mathit{sessionId} \equiv \mathit{reference}[\texttt{session}]$.

■

Lemma 6 (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 77) to p in Q, then t was *created* (Definition 75) 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 17. $\mathsf{signkey}(as) \equiv S(as).\texttt{jwk}$ follows from Definition 17 and by induction over the processing steps: state subterm `jwk` of a client is never changed.

By Definitions 16, 17, 18, 60, and Appendix D-C, 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 16, 17, 18, 7, and Appendix D-C, 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 51 and Definition 75, 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 70), e must have been *emitted* by some process in a processing step s_e prior to Q' in ρ. Definition 51 and Definition 72 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 7 (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 $\mathit{clientId}$ at as, every $i \in \mathbb{N}$ with $0 \leq i \leq |S(as).\texttt{mtlsRequests}[\mathit{clientId}]|$, and every process p with $as \neq p \neq c$ it holds true that $\mathit{mtlsNonce} := S(as).\texttt{mtlsRequests}[\mathit{clientId}].i.1$ does not leak to p, i.e., $\mathit{mtlsNonce} \notin d_\emptyset(S^n(p))$.

PROOF. Initially, the `mtlsRequests` subterm of the authorization server's state is empty, i.e., $S^0(as).\texttt{mtlsRequests} \equiv \langle\rangle$ (Definition 17). 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 $(S^i, E^i, N^i) \rightarrow (S^{i'}, E^{i'}, N^{i'})$ 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 authorization server sends out the nonce in Line 163 of Algorithm 9, asymmetrically encrypted with the public key

$$
\begin{aligned}
& clientKey \\
\equiv\ & S^i(as).\texttt{clients}[clientId][\texttt{mtls\_key}] && \text{(Line 158, Algorithm 9)} \\
\equiv\ & s_0^{as}.\texttt{clients}[clientId][\texttt{mtls\_key}] && \text{(value is never changed)} \\
\equiv\ & \mathsf{clientInfoAS}(as)[clientId][\texttt{mtls\_key}] && \text{(Definition 17)} \\
\equiv\ & \langle\{\langle cli.\texttt{client\_id}, \mathsf{as\_cli}(cli)\rangle \mid d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as)\}\rangle[clientId][\texttt{mtls\_key}] && \text{(Definition 14)}
\end{aligned}
$$

with $cli = \mathsf{clientInfo}(d_{as}, d_c)$ and $\mathsf{as\_cli}$ as in Definition 14. As $clientId$ is the identifier of c at as and as no two clients have the same identifier at an authorization server (see Definition 12), 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

$$
\begin{aligned}
& clientKey \\
\equiv\ & \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{tlsSKey}) && (d_{as} \in \mathsf{dom}(as),\ d_c \in \mathsf{dom}(c);\ \text{see also Definition 17}) \\
\equiv\ & \mathsf{pub}(\mathsf{tlskey}(d_c)) && (d_c \in \mathsf{dom}(c))
\end{aligned}
$$

The corresponding private key is $\mathsf{tlskey}(d_c) \in tlskeys^c$, which is only known to c (Lemma 3). (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 in Line 3 of Algorithm 2, the client stores the sequence $\langle request.\texttt{host}, clientId, pubKey, mtlsNonce\rangle$ into the `mtlsCache` subterm of its state in Line 8 of Algorithm 2, where $clientId, pubKey \in \mathcal{T}_{\mathcal{N}}$ and, in particular,

- $request.\texttt{host}$ 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 entry of the sequence containing the mTLS nonce (stored in the `mtlsCache` subterm). Let $req_{c \to as}$ be the HTTP request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c \to as}$ (which contains the mTLS nonce) in the `pendingDNS` state subterm of c, see Line 2 of Algorithm 26 and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 26. Thus, after finishing this processing step, the client stores the mTLS nonce only in its `pendingDNS` state subterm.

The client acceses the `pendingDNS` state subterm only within the else case in Line 10 of Algorithm 31, i.e., when it receives the DNS response. There, it either stops without a new event and without changing its state in Line 12 of Algorithm 31, or creates a new `pendingRequests` entry containing the request $req_{c \to as}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 31. In this case, the client removes the request from the `pendingDNS` state subterm in Line 17 of Algorithm 31, i.e., regarding the client state, the mTLS nonce is only contained in the newly created `pendingRequests` entry. The client finishes the processing step by encrypting $req_{c \to as}$ with the key of the domain that was stored along with the mTLS nonce, i.e., a key of as, see Lines 16 and 18 of Algorithm 31.

as does not leak mTLS nonce contained in request.

As the client encrypts $req_{c \to as}$ asymetrically with a key of as, it follows that only as can decrypt the HTTPS request (Lemma 27).

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 the client receives the HTTPS response to $req_{c \to as}$, 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 request, we conclude that the nonce never leaks to any other process, in particular not to p. ■

Lemma 8 (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 7:

Initially, the `mtlsRequests` subterm of the resource server's state is empty, i.e., $S^0(rs).\texttt{mtlsRequests} \equiv \langle\rangle$ (Definition 18). 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 $(S^i, E^i, N^i) \to (S^{i'}, E^{i'}, N^{i'})$ 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 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 3). (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 in Line 3 of Algorithm 2, the client stores the sequence $\langle request.\texttt{host}, clientId, pubKey, mtlsNonce\rangle$ into the `mtlsCache` subterm of its state, where $clientId, pubKey \in \mathcal{T}_{\mathcal{N}}$ and, 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 entry of the sequence containing the mTLS nonce (stored in the `mtlsCache` subterm). Let $req_{c \to rs}$ be the request that the client sends by calling HTTPS_SIMPLE_SEND.

HTTPS_SIMPLE_SEND stores the request $req_{c \to rs}$ (which contains the mTLS nonce) in the `pendingDNS` state subterm of c, see Line 2 of Algorithm 26, and then stops with the DNS request (which does not contain the nonce) in Line 3 of Algorithm 26. Thus, after finishing this processing step, the client stores the mTLS nonce only in its `pendingDNS` state subterm.

The client acceses the `pendingDNS` state subterm only within the else case in Line 10 of Algorithm 31, i.e., when it receives the DNS response. There, it either stops without a new event and without changing its state in Line 12 of Algorithm 31, or creates a new `pendingRequests` entry containing the request $req_{c \to rs}$ (and thus, also the mTLS nonce) in Line 15 of Algorithm 31. In this case, the client removes the request from the `pendingDNS` state subterm in Line 17 of Algorithm 31, i.e., regarding the client state, the mTLS nonce is only contained in the newly created `pendingRequests` entry. The client finishes the processing step by encrypting $req_{c \to rs}$ with the key of the domain that was stored along with the mTLS nonce, i.e., a key of rs, see Lines 16 and 18 of Algorithm 31.

rs does not leak mTLS nonce contained in request. As the HTTP request $req_{c \to rs}$ is is encrypted asymetrically with a key of rs, it follows that only the resource server can decrypt the request. 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 response to $req_{c \to rs}$, 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 9 (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 ($s^c_0.\texttt{jwk}$, see Definition 16). Initially, no other process and no waiting event contains the key except as a public key $\mathsf{pub}(\mathsf{signkey}(c))$, which cannot be used to retrieve the private key.

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 (Lines 19, 35, and 37), Algorithm 4 (Lines 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 that HTTP request (the token request) to the token endpoint it has cached for the AS identified by the issuer identifier in $\mathsf{extractmsg}(t)[\texttt{aud}]$ (i.e., $selectedAS$ in the context of Algorithm 3). From Lemma 16, we know that this token endpoint is a URL of the selected AS, i.e., the token request is sent to and encrypted for the party to which the domain $selectedAS$ belongs (see the call of HTTPS_SIMPLE_SEND in Line 41 (using `responseTo: TOKEN` in the first function argument). This party is an honest AS by the preconditions of this lemma.

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). Similar to the first case, this request (the PAR request) is encrypted for and sent to the PAR endpoint c has cached for the party to which $\mathsf{extractmsg}(t)[\texttt{aud}]$ belongs. Analogous to the first case, we can apply Lemma 16 to conclude that this party must be an honest AS (and the request is stored by c with `responseTo: PAR`).

When the client receives a corresponding 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 and hence cannot leak it in any way.

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 10 (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 75):

- $\texttt{client\_id} \in m.\texttt{body} \Rightarrow m.\texttt{body}[\texttt{client\_id}] \equiv clientId$ 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 27 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

$$
\begin{aligned}
& S(as).\texttt{clients}[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Line 4 and 12, Algorithm 10)} \\
\equiv\ & s_0^{as}.\texttt{clients}[\mathit{clientId}][\texttt{jwk\_key}] && \text{(value is never changed)} \\
\equiv\ & \mathsf{clientInfoAS}(as)[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Definition 17)} \\
\equiv\ & \langle\{\langle \mathit{cli}.\texttt{client\_id}, \mathsf{as\_cli}(\mathit{cli})\rangle \mid d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as)\}\rangle[\mathit{clientId}][\texttt{jwk\_key}] && \text{(Definition 14)}
\end{aligned}
$$

with $\mathit{cli} = \mathsf{clientInfo}(d_{as}, d_c)$ and $\mathsf{as\_cli}$ as in Definition 14. 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 12), 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 $\texttt{jwk\_key}$ (otherwise, as would **stop** with empty E_{out} in Line 5 of Algorithm 10), it follows that the verification key is

$$
\begin{aligned}
& \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{jwtSkey}) && (d_{as} \in \mathsf{dom}(as),\ d_c \in \mathsf{dom}(c);\ \text{see also Definition 14}) \\
\equiv\ & \mathsf{pub}(\mathsf{signkey}(c)) && \text{(Definition 12)}
\end{aligned}
$$

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 9.

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 7 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 11 (mTLS Keys of Clients stored at Authorization Servers). For every authorization server $as \in \mathsf{AS}$ and every term $\mathit{clientId} \in \mathcal{T}_{\mathcal{N}}$ it holds true that if $s_0^{as}.\texttt{clients}[\mathit{clientId}][\texttt{mtls\_key}]$ is not $\mathsf{pub}(\diamond)$ and not $\langle\rangle$, then $\exists c \in \mathsf{C}, d \in \mathsf{dom}(c)$ such that $s_0^{as}.\texttt{clients}[\mathit{clientId}][\texttt{mtls\_key}] \equiv \mathsf{pub}(\mathsf{tlskey}(d))$.

PROOF.

$$
\begin{aligned}
& s_0^{as}.\texttt{clients}[\mathit{clientId}][\texttt{mtls\_key}] && (1) \\
\equiv\ & \mathsf{clientInfoAS}(as)[\mathit{clientId}][\texttt{mtls\_key}] && (2) \\
\equiv\ & \langle\{\langle \mathit{cli}.\texttt{client\_id}, \mathsf{as\_cli}(\mathit{cli})\rangle \mid \exists\, d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as) \colon \mathsf{clientInfo}(d_{as}, d_c) \equiv \mathit{cli}\}\rangle[\mathit{clientId}][\texttt{mtls\_key}] && (3) \\
\equiv\ & \mathsf{as\_cli}(\mathsf{clientInfo}(d_{as}, d_c))[\texttt{mtls\_key}]\ (\text{with } d_c \in \mathsf{Doms}, d_{as} \in \mathsf{dom}(as), \mathsf{clientInfo}(d_{as}, d_c)[\texttt{client\_id}] \equiv \mathit{clientId}) && (4) \\
\equiv\ & \mathsf{pub}(\mathsf{clientInfo}(d_{as}, d_c).\texttt{mtlsSKey}) && (5)
\end{aligned}
$$

Notes on the above equivalences:

(2) As per Definition 17.

(3) See Definition 14.

(5) The value is not $\langle\rangle$ (precondition); with $d_c \in \mathsf{Doms}$, $d_{as} \in \mathsf{dom}(as)$, $\mathsf{clientInfo}(d_{as}, d_c)[\texttt{client\_id}] \equiv \mathit{clientId}$; see Definition 14 for the definition of as_cli.

As this value is not $\mathsf{pub}(\diamond)$ (precondition of the lemma), it follows from Definition 12 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 12).

Lemma 12 (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. From Lemma 2, we have that 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 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 Lines 27 and 29 of Algorithm 4.

We note that the generic part of the client model (which takes care of DNS resolution and sending the actual HTTPS request after the HTTPS_SIMPLE_SEND call) does not send out or use t in any way – except for the sending of the actual request, which is encrypted for the domain $\mathsf{extractmsg}(t)[\texttt{payload}][\texttt{htu}].\texttt{host}$, i.e., for rs, which can only be decrypted by rs (Lemma 27).

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 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 Lines 19 and 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 13 (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 $\exists r, x, y, k, m_{\text{resp}}.\ \langle x, y, \mathsf{enc_s}(m_{\text{resp}}, k)\rangle \in^{\langle\rangle} E^Q_{\text{out}}$ such that m_{resp} is an HTTP response, $r := m_{\text{resp}}.\texttt{body}[\texttt{resource}]$, and $r \in^{\langle\rangle} S^{Q\prime}(rs).\texttt{resourceNonce}[id][resourceID]$, 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 is a fresh nonce generated in P at the resource endpoint of rs 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 1).
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

$$\begin{aligned}
& S^Q(rs).\texttt{resourceASMapping}[resourceID] && \text{(Line 15 of Algorithm 12)}\\
\equiv\ & s^{rs}_0.\texttt{resourceASMapping}[resourceID] && \text{(value is never changed)}\\
\in\ & \mathsf{dom}(\mathsf{authorizationServerOfResource}^{rs}(resourceID)) && \text{(Definition 18)}
\end{aligned}$$

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

$$\begin{aligned}
& asInfo[\texttt{as_key}] && \\
\equiv\ & S^Q(rs).\texttt{asInfo}[responsibleAS][\texttt{as_key}] && (responsibleAS \in \mathsf{dom}(as), \text{Line 18})\\
\equiv\ & s^{rs}_0.\texttt{asInfo}[responsibleAS][\texttt{as_key}] && \text{(value is never changed)}\\
\equiv\ & \mathsf{signkey}(\mathsf{dom}^{-1}(responsibleAS)) && \text{(Definition 18)}\\
\equiv\ & \mathsf{signkey}(as) &&
\end{aligned}$$

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). Note: we know that i is a natural number and not a "longer" pointer due to the last condition in Line 92 of Algorithm 9.

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 12 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 12 and conclude that `dpopProof` can only be known by c and rs. The only places where a resource server sends a request are Lines 55 and 67 of Algorithm 12. In the first case, the request in question is a token introspection request whose Authorization header uses the `Basic` scheme. Processing of such a request by the resource server would lead to an empty E^Q_{out} in Line 45 of Algorithm 12. In the latter case, the resource server leaks the resource request – but only after invalidating the mTLS nonce (Lines 24f. of Algorithm 12) or DPoP nonce (Line 42 of Algorithm 12), i.e., processing this request again would lead to an empty E^Q_{out} in Line 24 of Algorithm 12, or Line 38 of Algorithm 12. Hence, resource servers do not send requests with valid DPoP or mTLS nonces to themselves and 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 17). 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 11 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 $\mathsf{tlskey}(d_c)$ (Lemma 3), 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 $\mathsf{signkey}(\mathsf{c})$ (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 16).
- 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 8 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 18).

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_{in}^{P}.

■

Lemma 14 (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 $rec[\texttt{redirect\_uri}].\texttt{protocol} \equiv \mathsf{S}$, 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 10.

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 10, 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 15 (Integrity of Client's Session Storage). 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 login session id $lsid$, we have that if $lsid \in S(c).\texttt{sessions}$, then all of the following hold true:

1) $\texttt{selected\_AS} \in S(c).\texttt{sessions}[lsid]$
2) for all configurations (S', E', N') after (S, E, N) in ρ we have $S'(c).\texttt{sessions}[lsid][\texttt{selected\_AS}] \equiv S(c).\texttt{sessions}[lsid][\texttt{selected\_AS}]$

PROOF. Since we have $S^0(c).\texttt{sessions}$ (Definition 16), we know that if $lsid \in S(c).\texttt{sessions}$, such an entry must have been stored there by c. Clients only ever store/add such an entry in Line 10 of Algorithm 1, where a key `selected_AS` is part of the stored entry. The key used to refer to the entry inside `sessions` is a fresh nonce (i.e., $lsid$ is a fresh nonce there). Hence, whenever a client first stores an entry in `sessions` under key $lsid$, this entry contains a key `selected_AS`.

It is easy to see that Line 10 of Algorithm 1 is indeed the only place in which a client stores any value under key `selected_AS` in the `sessions` state subterm. Similarly, it is easy to check that this line is also the only place in which a client (over)writes a whole entry in the `sessions` state subterm. Hence, we can conclude: A key `selected_AS` is present whenever a client adds an entry to the `sessions` state subterm and neither the value stored under that key, nor the `sessions` entry itself are overwritten or removed anywhere, implying 1). In addition, if the client ever executes Line 10 of Algorithm 1 again, it will never overwrite an existing entry, because it will use a fresh login session id, thus we have 2). ■

Lemma 16 (Integrity of Client's `oauthConfigCache`*).* 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, and every domain $d \in \mathsf{dom}(as)$, it holds true that if $d \in S(c).\texttt{oauthConfigCache}$, we have all of the following:

1) $S(c).\texttt{oauthConfigCache}[d][\texttt{issuer}] \equiv d$
2) $S(c).\texttt{oauthConfigCache}[d][\texttt{auth\_ep}] \equiv \langle \texttt{URL}, \mathsf{S}, d, /\texttt{auth}, \langle\rangle, \bot\rangle$

3) $S(c).\texttt{oauthConfigCache}[d][\texttt{token\_ep}] \equiv \langle \texttt{URL}, \texttt{S}, d, /\texttt{token}, \langle\rangle, \bot \rangle$
4) $S(c).\texttt{oauthConfigCache}[d][\texttt{par\_ep}] \equiv \langle \texttt{URL}, \texttt{S}, d, /\texttt{par}, \langle\rangle, \bot \rangle$
5) $S(c).\texttt{oauthConfigCache}[d][\texttt{introspec\_ep}] \equiv \langle \texttt{URL}, \texttt{S}, d, /\texttt{introspection}, \langle\rangle, \bot \rangle$
6) $S(c).\texttt{oauthConfigCache}[d][\texttt{jwks\_uri}] \equiv \langle \texttt{URL}, \texttt{S}, d, /\texttt{jwks}, \langle\rangle, \bot \rangle$

We note that this implies that all these entries in $S(c).\texttt{oauthConfigCache}[d]$ are never changed once they have been stored and that all entries are created in the same processing step.

PROOF. We start by noting that $S^0(c).\texttt{oauthConfigCache} \equiv \langle\rangle$ (Definition 16), i.e., the `oauthConfigCache` state subterm is initially empty. An honest client only ever writes to its `oauthConfigCache` state subterm in Line 16 of Algorithm 2 when processing an HTTPS response. Hence, $d \in S(c).\texttt{oauthConfigCache}$ implies that there must have been a processing step $Q = (S^Q, E^Q, N^Q) \to (S^{Q'}, E^{Q'}, N^{Q'})$ in ρ such that $d \notin S^Q(c).\texttt{oauthConfigCache}$ and $d \in S^{Q'}(c).\texttt{oauthConfigCache}$. In Q, PROCESS_HTTPS_RESPONSE must have been called with a *reference* as second argument, such that $\mathit{reference}[\texttt{responseTo}] \equiv \texttt{CONFIG}$. In addition, $\mathit{reference}[\texttt{session}]$ must contain a value *sessionId* such that $S^Q(c).\texttt{sessions}[\mathit{sessionId}][\texttt{selected\_AS}] \equiv m.\texttt{body}[\texttt{issuer}]$ (Line 14 of Algorithm 2). From Line 16 of Algorithm 2, we also know that $S^Q(c).\texttt{sessions}[\mathit{sessionId}][\texttt{selected\_AS}] \equiv d$ (cf. Lemma 15). Hence, we already have that $d \in S(c).\texttt{oauthConfigCache}$ implies 1).

With Lemma 4, we have that there must be a processing step $P = (S^P, E^P, N^P) \to (S^{P'}, E^{P'}, N^{P'})$ prior to Q in ρ in which c called HTTPS_SIMPLE_SEND with *reference* as first argument. Such a *reference* (one with `responseTo` set to `CONFIG`) is only created in Line 9 of Algorithm 6. The accompanying message's host value there is $S^P(c).\texttt{sessions}[\mathit{sessionId}][\texttt{selected\_AS}]$, i.e., by Lemma 15, d. That same message's path value is either $/.\texttt{well\_known}/\texttt{openid-configuration}$ or $/.\texttt{well\_known}/\texttt{oauth-authorization-server}$. From Lemma 27, Algorithm 26, and Lines 10ff. of Algorithm 31 (and because as does not leak $\mathsf{tlskey}(d)$), we know that the request given to HTTPS_SIMPLE_SEND in P can only be answered by as (and c, but clients do not reply to requests with the aforementioned path values).

Such a request, i.e., one with the path values mentioned above, is processed by as in Lines 2ff. of Algorithm 9. From looking at those Lines, it is obvious that the response sent in Line 10 of Algorithm 9 contains a body with a dictionary fulfilling 2)–6). Using Lemma 27 once more, we can conclude that c processes such a response in Q and thus we have 2)–6). ■

Lemma 17 (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 $\mathit{code} \not\equiv \bot$ for which there is a record $\mathit{rec} \in^{\langle\rangle} S(as).\texttt{records}$ with $\mathit{rec}[\texttt{code}] \equiv \mathit{code}$, $\mathit{rec}[\texttt{client\_id}] \equiv \mathit{clientId}$, and $\mathit{rec}[\texttt{subject}] \equiv id$ and every process $p \notin \{as, c, b\}$, it holds true that $\mathit{code} \notin d_\emptyset(S(p))$.

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 17 – `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 $/\texttt{auth2}$ 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 $\mathit{password} \equiv \mathsf{secretOfID}(\mathit{identity})$. Since as does not send requests to itself and $\mathsf{secretOfID}(\mathit{identity})$ is only known to as and $\mathsf{ownerOfID}(\mathit{identity})$, *req* must have been created by $\mathsf{ownerOfID}(\mathit{identity})$ if the sender of *req* is honest. W.l.o.g., let $\mathit{identity} \equiv id$, 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 $S(as).\texttt{authorizationRequests}[\mathit{requestUri}][\texttt{redirect\_uri}]$

where $S(as)$.authorizationRequests[$requestUri$][client_id] $\equiv$ $clientId$. With Lemma 14, 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 $\langle$__Host, 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, Line 16 of Algorithm 2, and Lemma 16), 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. With Lemma 27 we can conclude that 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 18 (Unique Code Verifier for Each Login Session ID at Client). 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^i, E^i, N^i) in ρ, every client $c \in \mathsf{C}$ that is honest in S with client identifier $clientId$ at as, every login session id $lsid$, and every term $codeVerifier$, we have that $S^i(c)$.sessions[$lsid$][code_verifier] $\equiv$ $codeVerifier$ implies:

1) $S^j(c)$.sessions[$lsid$][code_verifier] $\equiv$ $codeVerifier$ for all $j \geq i$, and
2) $S^i(c)$.sessions[$lsid'$][code_verifier] $\not\equiv$ $codeVerifier$ for all $lsid' \not\equiv lsid$.

PROOF. We start by noting that an honest client only ever stores something in an entry in sessions under key code_verifier in Line 34 of Algorithm 6. The value stored there is always a fresh nonce (see Line 26 of Algorithm 6). Hence, we can conclude 2).

To get 1), we need to prove that a stored code verifier is never overwritten. For this, we show that a client executes Line 34 of Algorithm 6 at most once with the same login session id (i.e., $sessionId$ in the context of said line). For this, we look at the places where Algorithm 6 (PREPARE_AND_SEND_PAR) is called. Note that the first argument to PREPARE_AND_SEND_PAR is the aforementioned $sessionId$:

Line 11 of Algorithm 1 Here, the first argument is a fresh nonce (see Line 9 of Algorithm 1), i.e., this line will never lead to PREPARE_AND_SEND_PAR being called a second time with a given $sessionId$.

Line 17 of Algorithm 2 This line is only executed when the client processes an HTTPS response such that Algorithm 2 (PROCESS_HTTPS_RESPONSE) was called with a $reference$ containing a key responseTo with value CONFIG. The $sessionId$ value used when calling PREPARE_AND_SEND_PAR is also taken from the $reference$ (see Line 10 of Algorithm 2). I.e., we have to check where this $reference$ came from.

$reference$ is one of the arguments to PROCESS_HTTPS_RESPONSE which is only called in Line 26 of Algorithm 31, where the value for $reference$ is taken from the client's pendingRequests state subterm. The pendingRequests state subterm is initially empty (Definition 16) and the only place where elements are added to this state subterm is Line 15 of Algorithm 31. There, in turn, the value for $reference$ is taken (unchanged) from an entry in the pendingDNS state subterm. Once again, this state subterm is initially empty and there is only one place in which entries are added to it: In Line 2 of Algorithm 26, i.e., in HTTPS_SIMPLE_SEND, where $reference$ is one of the arguments. Hence, we have to look at places where HTTPS_SIMPLE_SEND is called with a $reference$ where $reference$[responseTo] $\equiv$ CONFIG.

The only place where such a $reference$ is passed to HTTPS_SIMPLE_SEND is Line 9 of Algorithm 6. However, this call always ends in a **stop** and the call happens *before* the client executes Line 34 of Algorithm 6 – hence, if an execution of PREPARE_AND_SEND_PAR leads to execution of Line 11 of Algorithm 1 and thus a subsequent call of PREPARE_AND_SEND_PAR, both calls use the same $sessionId$, but Line 34 of Algorithm 6 (i.e., storing a code verifier) is executed at most once.

Line 20 of Algorithm 2 This case is very similar to the previous one, except for the following changes: The responseTo value in question is JWKS instead of CONFIG, and the place in which HTTPS_SIMPLE_SEND is called with a suitable $reference$ is Line 14 of Algorithm 6. ■

Lemma 19 (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 23),

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 (Lemma 16), 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) and Lemma 16. 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** $\mathsf{hash}(codeVerifier)$**.** 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`, see Lemma 18), 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 $\langle\langle$`__Host`, `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 2 holds.

Lemma 20 (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 $id \in^{\langle\rangle} s_0^{rs}$.`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 $\exists r, x, y, k, m_{\text{resp}}.\langle x, y, \mathsf{enc_s}(m_{\text{resp}}, k)\rangle \in^{\langle\rangle} E^Q_{\text{out}}$ such that m_{resp} is an HTTP response, $r := m_{\text{resp}}$.`body`[`resource`], and $r \in^{\langle\rangle} S^{Q\prime}(rs)$.`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., $t \equiv m_{\text{req}}$.`headers`[`Authorization`].2, and
c) r is a fresh nonce generated in P at the resource endpoint of rs 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 1).
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 13, 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. This response can only be decrypted by c: A resource server can decrypt symmetrically only in Line 19 of Algorithm 31 (i.e., in the generic server model), where the decryption key is taken from the `pendingRequests` state subterm. The application-layer model of a resource server does not access this state subterm, and the generic HTTPS server model stores only fresh nonces as keys (see Line 15 of Algorithm 31).

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 (as a client sends out requests with an `Authorization` header only by calling HTTPS_SIMPLE_SEND in Line 30 of Algorithm 4 with the reference `RESOURCE_USAGE`) 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 16). 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 5, 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 17, 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 4 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 21 (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 3, 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 (one of) as's token endpoints (cf. Lemma 16). 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 10).

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 17. ■

D. Session Integrity for Authentication Property

In this section, we show that the authentication property from Definition 24 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 22 (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 19, 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 login session 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 13 and Definition 17).

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 19 (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 19, 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 25 holds.

Lemma 23 (Session Integrity for Authorization Property Holds). 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 $\mathsf{accessesResource}_\rho^{\mathsf{Q}}(b, r, u, c, rs, as, lsid)$ and c did not leak the authorization request for $lsid$ (see Definition 23), 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 `Set-Cookie` (with `sessionid`) only in Line 30 of Algorithm 2. The remaining proof is analogous

to the proof of the first postcondition of Lemma 22.

(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 20, 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 18).

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 16). 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 19 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 22.

■

F. Proof of Theorem

Theorem 1 follows from Lemma 20, Lemma 21, Lemma 22, and Lemma 23.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{6}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{7}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{8}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{9}$$
$$\mathsf{checkmac}(\mathsf{mac}(x, y), y) = \top \tag{10}$$
$$\mathsf{extractmsg}(\mathsf{mac}(x, y)) = x \tag{11}$$
$$\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i \text{ if } 1 \leq i \leq n \tag{12}$$
$$\pi_j(\langle x_1, \ldots, x_n \rangle) = \diamond \text{ if } j \notin \{1, \ldots, n\} \tag{13}$$
$$\pi_j(t) = \diamond \text{ if } t \text{ is not a sequence} \tag{14}$$

Figure 8. Equational theory for Σ.

APPENDIX G
TECHNICAL DEFINITIONS

Here, we provide technical definitions of the WIM. These follow the descriptions in [28, 33–38].

A. Terms and Notations

Definition 26 (Signature Σ). We define the signature Σ, over which we will define formal terms, as the union of the following pairwise disjoint sets:

Constants $\mathcal{C} = \mathbb{S} \cup \mathsf{IPs} \cup \{\bot, \top, \diamond\}$ with the three sets pairwise disjoint. $\mathbb{S}$is the set of all (ASCII) strings, including the empty string ε. IPsis the set of IP addresses.

Function Symbols to represent public keys, asymmetric encryption and decryption, symmetric encryption and decryption, signatures, signature verification, MACs, MAC verification, message extraction from signatures and MACs, and hashing, respectively: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, $\mathsf{mac}(\cdot,\cdot)$, $\mathsf{checkmac}(\cdot,\cdot)$, $\mathsf{extractmsg}(\cdot)$, $\mathsf{hash}(\cdot)$.

Sequences of any length $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc. Note that formally, these sequence "constructors" are also function symbols.

Projection Symbols to access sequence elements: $\pi_i(\cdot)$ for all $i \in \mathbb{N}_\emptyset$. Note that formally, projection symbols are also function symbols.

Definition 27 (Nonces and Terms). By $X = \{x_0, x_1, \ldots\}$ 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 $t \in N \cup X \cup \mathcal{C}$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol for some $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ 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 8). 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 28 (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 $V_{\text{process}} = \{\nu_1, \nu_2, \ldots\}$ 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 29 (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., $\langle\rangle$, $\langle\langle a, f, m\rangle, \langle a', f', m'\rangle, \ldots\rangle$, etc.).

Definition 30 (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 8. For a term t, we denote its normal form as $t\!\downarrow$.

Definition 31 (Pattern Matching). Let $pattern \in \mathcal{T}_\mathcal{N}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches* $pattern$ iff t can be acquired from $pattern$ by replacing each occurrence of the wildcard with an arbitrary term (which may

be different for each instance of the wildcard). We write $t \sim \mathit{pattern}$. For a sequence of patterns $\mathit{patterns}$ we write $t \mathrel{\dot\sim} \mathit{patterns}$ to denote that t matches at least one pattern in $\mathit{patterns}$.

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

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

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

Definition 32 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$, and $t_1, \ldots, t_n \in \mathcal{T}_N$.

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

Definition 33 (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 34 (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 35 (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 36 (Pointers). A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

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

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

Definition 37 (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 38 (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 39. 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 36):

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

If the $secure$ attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections.[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 [15]) of a cookie is set (i.e., $name$ consists of two parts and $name.1 \equiv$ __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 42. An *HTTP request* is a term of the form shown in (15). An *HTTP response* is a term of the form shown in (16).

$$\langle \texttt{HTTPReq}, nonce, method, host, path, parameters, headers, body \rangle \tag{15}$$

$$\langle \texttt{HTTPResp}, nonce, status, headers, body \rangle \tag{16}$$

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.

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

- $host \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $path \in \mathbb{S}$ indicates the resource path at the server side.
- $status \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard).
- $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters.
- $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \texttt{Origin}, o \rangle$ where o is an origin,
  - $\langle \texttt{Set-Cookie}, c \rangle$ where c is a sequence of cookies,
  - $\langle \texttt{Cookie}, c \rangle$ where $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \texttt{Location}, l \rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \texttt{Referer}, r \rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \texttt{Strict-Transport-Security}, \top \rangle$,
  - $\langle \texttt{Authorization}, \langle username, password \rangle \rangle$ where $username, password \in \mathbb{S}$ (this header models the 'Basic' HTTP Authentication Scheme, see [67]),
  - $\langle \texttt{ReferrerPolicy}, p \rangle$ where $p \in \{\texttt{noreferrer}, \texttt{origin}\}$.
- $body \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

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

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, \texttt{1} \rangle\rangle, [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S} \rangle], \langle \texttt{foo}, \texttt{bar} \rangle \rangle \tag{17}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top \rangle\rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x \rangle \rangle \tag{18}$$

An HTTP `POST` request for the URL http://example.com/show?index=1 is shown in (17), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar} \rangle$. A possible response is shown in (18), 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 G-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 43. 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 $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

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

Example 7.

$$\mathsf{enc_a}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{19}$$

$$\mathsf{enc_s}(s, k') \tag{20}$$

The term (19) shows an encrypted request (with r as in (17)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (20) is a response (with s as in (18)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (19).

5) DNS Messages:

Definition 44. 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 $\mathsf{DNSRequests}$.

Definition 45. 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 $\mathsf{DNSResponses}$.

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

C. Atomic Processes, Systems and Runs

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

Definition 46 (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 47 (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 48 (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) $E_{\text{out}}^\nu \subseteq 2^{\mathcal{E}^\nu\langle\rangle}$ of protoevents,
2) a term $s^\nu \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3) a sequence $(v_1, v_2, \dots, v_i)$ of all placeholders appearing in E_{out}^ν (ordered lexicographically),
4) a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ of the first i elements in N

with

1) $((e_{\text{in}}, S(p)), (E_{\text{out}}^\nu, s^\nu)) \in R^p$ and $a \in I^p$,
2) $E_{\text{out}} = E_{\text{out}}^\nu[\eta_1/v_1, \dots, \eta_i/v_i]$,
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 49 (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 $S^0(p) = s_0^p$ 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 50 (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., $a \in d_\emptyset(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$.

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

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 51 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that p is an atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, with $((e,s),(E,s')) \in R^p$ it holds true that $E, s' \in d_{V_{\text{process}}}(\{e,s\})$.

E. Attackers

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

Definition 52 (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. If N is empty, the corresponding processing step in which this selection happens does not finish. We write **for** $s \in M$ **do** to denote that the following commands 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 we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 27). Table I shows a list of some of the placeholders, generally denoted by ν with some subscript to distinguish between multiple fresh values.

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.

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
$\nu_{10}, \dots$	Algorithm 20, replacement for placeholders in script output

Table I: List of placeholders used in browser algorithms.

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

Definition 54 (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 55 (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 56. 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 57. 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, $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $referrer \in \text{URLs} \cup \{\bot\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\langle\rangle}$ 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 58. 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 59. 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 34.

Definition 60. The *set of states $Z_{\text{webbrowser}}$ of a web browser atomic Dolev-Yao process* consists of the terms of the form

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

with the subterms as follows:

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

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 61. 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.

[10]Recall the definition of a pointer in Definition 36.

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 62. 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.)

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 63. 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 $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ 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 $\mathsf{CookieMerge}(oldcookies, newcookies, protocol)$ is m.

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

Definition 64. 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 $\mathsf{AddCookie}(oldcookies, c, protocol)$ 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 [7], Section 5.1.4 for this definition.

Definition 65. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w'}.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists\, \overline{w''} \in \mathsf{Subwindows}(s')$ with $s'.\overline{w'} \xrightarrow{\text{childof}^*} s'.\overline{w''}$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\, s'.\overline{p}.\texttt{activedocument.origin} = s'.\overline{w}.\texttt{activedocument.origin}$ ($\overline{w'}$ 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 $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ 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':
  - 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.

Algorithm 14 Web Browser Model: Determine window for navigation.

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

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 $referrerPolicy \equiv$ `no-referrer` **then**
    let $referrer := \bot$
  if $referrer \not\equiv \bot$ **then**
    if $referrerPolicy \equiv$ `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 $\langle \langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, \nu_8 \rangle \rangle, s'$

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** $s'.\overline{w'}.\texttt{documents}.\bar{j}.\texttt{active} \equiv \top$
    $\hookrightarrow \ \wedge\ s'.\overline{w'}.\texttt{documents}.(\bar{j}+1) \in \mathsf{Documents}$ **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 $s' := \mathsf{CANCELNAV}(s'.\overline{w'}.\texttt{nonce}, s')$
  stop $\langle\rangle, s'$

Algorithm 20 Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, a, s')
  let $tree := \mathsf{Clean}(s', s'.\overline{d})$
  let $cookies := \langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
    $\hookrightarrow \ \wedge\ c.\texttt{content.httpOnly} \equiv \bot$
    $\hookrightarrow \ \wedge\ \left(c.\texttt{content.secure} \equiv \top \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \mathsf{S}\right)\right)\}\rangle$
  let $tlw \leftarrow s'.\texttt{windows}$ **such that** tlw is the top-level window containing $\overline{d}$
  let $sessionStorage := s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
  let $localStorage := s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $secrets := s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $R := \mathsf{script}^{-1}(s'.\overline{d}.\texttt{script})$ **if possible; otherwise stop**
  let $in := \langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, cookies,$
    $\hookrightarrow \ localStorage, sessionStorage, s'.\texttt{ids}, secrets\rangle$
  let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, $cookies' \leftarrow \mathsf{Cookies}^{\nu}$, $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
    $\hookrightarrow \ sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, $command \leftarrow \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
    $\hookrightarrow \ out := \langle state', cookies', localStorage', sessionStorage', command\rangle$
    $\hookrightarrow \ $ **such that** $out := out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$ **with** $(in, out^{\lambda}) \in R$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right] :=$
    $\hookrightarrow \ \langle\mathsf{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], cookies', s'.\overline{d}.\texttt{origin.protocol})\rangle$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right] := localStorage'$
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right] := sessionStorage'$
  let $s'.\overline{d}.\texttt{scriptstate} := state'$
  let $referrer := s'.\overline{d}.\texttt{location}$
  let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
  let $docorigin := s'.\overline{d}.\texttt{origin}$
  switch $command$ **do**
    case $\langle\texttt{HREF}, url, hrefwindow, noreferrer\rangle$
      let $\overline{w'} := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, noreferrer, s')$
      let $reference := \langle\texttt{REQ}, s'.\overline{w'}.\texttt{nonce}\rangle$
      let $req := \langle\texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      if $noreferrer \equiv \top$ **then**
        let $referrerPolicy := \texttt{noreferrer}$
      let $s' := \mathsf{CANCELNAV}(reference, s')$
      call HTTP_SEND($reference$, req, url, $\bot$, $referrer$, $referrerPolicy$, a, s')
    case $\langle\texttt{IFRAME}, url, window\rangle$
      if $window \equiv$ _SELF **then**
        let $\overline{w'} := \overline{w}$
      else
        let $\overline{w'} := \mathsf{GETWINDOW}(\overline{w}, window, s')$
      let $req := \langle\texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      let $w' := \langle\nu_5, \langle\rangle, \bot\rangle$
      let $s'.\overline{w'}.\texttt{activedocument.subwindows} := s'.\overline{w'}.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
      call HTTP_SEND($\langle\texttt{REQ}, \nu_5\rangle$, req, url, $\bot$, $referrer$, $referrerPolicy$, a, s')

→ Algorithm continues on next page.

```
36:         case ⟨FORM, url, method, data, hrefwindow⟩
37:             if method ∉ {GET, POST} then
38:                 stop
39:             let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
40:             let reference := ⟨REQ, s′.w̄′.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 ⟨CIBAFORM, url, method, data, hrefwindow, clientDomain, cibaBindingMessage⟩
                → Custom CIBA FORM command: When starting a CIBA flow, the client returns a binding message. When authenticating at the
                  AS, the end-user has to make sure that they receive the same value. For modeling this behavior, we extend the browser state
                  by the cibaBindingMessages subterm and define this command which first checks if the cibaBindingMessage is stored
                  by the browser and then continues as the FORM command. Note that this command is a modeling artifact.
53:             if ⟨clientDomain, cibaBindingMessage⟩ ∉^⟨⟩ s′.cibaBindingMessages then
54:                 stop
55:             if method ∉ {GET, POST} then
56:                 stop
57:             let w̄′ := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s′)
58:             let reference := ⟨REQ, s′.w̄′.nonce⟩
59:             if method = GET then
60:                 let body := ⟨⟩
61:                 let parameters := data
62:                 let origin := ⊥
63:             else
64:                 let body := data
65:                 let parameters := url.parameters
66:                 let origin := docorigin
67:             let req := ⟨HTTPReq, ν4, method, url.host, url.path, parameters, ⟨⟩, body⟩
68:             let s′ := CANCELNAV(reference, s′)
69:             call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
70:         case ⟨SETSCRIPT, window, script⟩
71:             let w̄′ := GETWINDOW(w̄, window, s′)
72:             let s′.w̄′.activedocument.script := script
73:             stop ⟨⟩, s′
74:         case ⟨SETSCRIPTSTATE, window, scriptstate⟩
75:             let w̄′ := GETWINDOW(w̄, window, s′)
76:             let s′.w̄′.activedocument.scriptstate := scriptstate
77:             stop ⟨⟩, s′
78:         case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
79:             if method ∈ {CONNECT, TRACE, TRACK} ∨ xhrreference ∉ V_process ∪ {⊥} then
80:                 stop
81:             if url.host ≢ docorigin.host ∨ url.protocol ≢ docorigin.protocol then
82:                 stop
83:             if method ∈ {GET, HEAD} then
84:                 let data := ⟨⟩
85:                 let origin := ⊥
86:             else
87:                 let origin := docorigin
88:             let req := ⟨HTTPReq, ν4, method, url.host, url.path, url.parameters, ⟨⟩, data⟩
89:             let reference := ⟨XHR, s′.d̄.nonce, xhrreference⟩
90:             call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
91:         case ⟨BACK, window⟩
92:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
93:             call NAVBACK(w̄′, s′)
```

→ Algorithm continues on next page.

```
94:     case ⟨FORWARD, window⟩
95:         let w' := GETNAVIGABLEWINDOW(w, window, ⊥, s')
96:         call NAVFORWARD(w', s')
97:     case ⟨CLOSE, window⟩
98:         let w' := GETNAVIGABLEWINDOW(w, window, ⊥, s')
99:         remove s'.w' from the sequence containing it
100:        stop ⟨⟩, s'
101:    case ⟨POSTMESSAGE, window, message, origin⟩
102:        let w' ← SUBWINDOWS(s') such that s'.w'.nonce ≡ window
103:        if ∃j ∈ ℕ such that s'.w'.documents.j.active ≡ ⊤
            ↪ ∧ (origin ≢ ⊥ ⟹ s'.w'.documents.j.origin ≡ origin) then
104:            let s'.w'.documents.j.scriptinputs := s'.w'.documents.j.scriptinputs
                ↪ +^⟨⟩ ⟨POSTMESSAGE, s'.w.nonce, docorigin, message⟩
105:        stop ⟨⟩, s'
106:    case else
107:        stop
```

- 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 59. *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. 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 59. *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 66. 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 67 (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 $s_0^p \in Z_{\text{webbrowser}}$.

Definition 68 (Web Browser Initial State). An initial state $s_0^p \in Z_{\text{webbrowser}}$ for a browser process p is a web browser state (Definition 60) with the following properties:

- s_0^p.windows $\equiv \langle\rangle$
- s_0^p.ids $\subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- s_0^p.secrets $\in [\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- s_0^p.cookies $\equiv \langle\rangle$
- s_0^p.localStorage $\equiv \langle\rangle$
- s_0^p.sessionStorage $\equiv \langle\rangle$

Algorithm 21 Web Browser Model: Process an HTTP response.

function PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, a, f, s')
  if `Set-Cookie` $\in$ $response$.`headers` **then**
    for each $c \in^{\langle\rangle}$ $response$.`headers` [`Set-Cookie`], $c \in$ Cookies **do**
      let s'.`cookies` [$request$.`host`] $\hookrightarrow$:= AddCookie(s'.`cookies` [$request$.`host`], c, $requestUrl$.`protocol`)
  if `Strict-Transport-Security` $\in$ $response$.`headers` $\wedge$ $requestUrl$.`protocol` $\equiv$ `S` **then**
    let s'.`sts` := s'.`sts` $+^{\langle\rangle}$ $request$.`host`
  if `Referer` $\in$ $request$.`headers` **then**
    let $referrer$:= $request$.`headers`[`Referer`]
  else
    let $referrer$:= $\bot$
  if `Location` $\in$ $response$.`headers` $\wedge$ $response$.`status` $\in \{303, 307\}$ **then**
    let url := $response$.`headers` [`Location`]
    if url.`fragment` $\equiv \bot$ **then**
      let url.`fragment` := $requestUrl$.`fragment`
    let $method'$:= $request$.`method`
    let $body'$:= $request$.`body`
    if `Origin` $\in$ $request$.`headers` $\hookrightarrow$ $\wedge$ $request$.`headers`[`Origin`] $\neq \diamond$ $\hookrightarrow$ $\wedge$ ($\langle url$.`host`, url.`protocol`$\rangle \equiv \langle request$.`host`, $requestUrl$.`protocol`$\rangle$ $\hookrightarrow$ $\vee$ $\langle request$.`host`, $requestUrl$.`protocol`$\rangle \equiv request$.`headers`[`Origin`]) **then**
      let $origin$:= $request$.`headers`[`Origin`]
    else
      let $origin$:= $\diamond$
    if $response$.`status` $\equiv 303 \wedge$ $request$.`method` $\notin \{$`GET`, `HEAD`$\}$ **then**
      let $method'$:= `GET`
      let $body'$:= $\langle\rangle$
    if $\exists\, \overline{w} \in$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv \pi_2(reference)$ **then** → Do not redirect XHRs.
      let req := $\langle$`HTTPReq`, ν_6, $method'$, url.`host`, url.`path`, url.`parameters`, $\langle\rangle$, $body'\rangle$
      let $referrerPolicy$:= $response$.`headers`[`ReferrerPolicy`]
      call HTTP_SEND($reference$, req, url, $origin$, $referrer$, $referrerPolicy$, a, s')
    else
      stop $\langle\rangle$, s'
  switch $\pi_1(reference)$ **do**
    case `REQ`
      let $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv \pi_2(reference)$ **if possible; otherwise stop** → normal response
      if $response$.`body` $\not\sim \langle *, * \rangle$ **then**
        stop $\langle\rangle$, s'
      let $script$:= $\pi_1(response$.`body`$)$
      let $scriptstate$:= $\pi_2(response$.`body`$)$
      let d := $\langle \nu_7, requestUrl, response$.`headers`, $referrer$, $script$, $scriptstate$, $\langle\rangle$, $\langle\rangle$, $\top\rangle$
      if $s'.\overline{w}$.`documents` $\equiv \langle\rangle$ **then**
        let $s'.\overline{w}$.`documents` := $\langle d \rangle$
      else
        let $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}$.`documents`.$\overline{i}$.`active` $\equiv \top$
        let $s'.\overline{w}$.`documents`.$\overline{i}$.`active` := $\bot$
        remove $s'.\overline{w}$.`documents`.$(\overline{i}+1)$ and all following documents $\hookrightarrow$ from $s'.\overline{w}$.`documents`
        let $s'.\overline{w}$.`documents` := $s'.\overline{w}$.`documents` $+^{\langle\rangle}$ d
      stop $\langle\rangle$, s'
    case `XHR`
      let $\overline{w} \leftarrow$ Subwindows(s'), $\overline{d}$ **such that** $s'.\overline{d}$.`nonce` $\equiv \pi_2(reference)$ $\hookrightarrow$ $\wedge\, s'.\overline{d} = s'.\overline{w}$.`activedocument` **if possible; otherwise stop** → process XHR response
      let $headers$:= $response$.`headers` $-$ `Set-Cookie`
      let $s'.\overline{d}$.`scriptinputs` := $s'.\overline{d}$.`scriptinputs` $+^{\langle\rangle}$ $\langle$`XMLHTTPREQUEST`, $headers$, $response$.`body`, $\pi_3(reference)\rangle$
      stop $\langle\rangle$, s'

- $s_0^p.\texttt{keyMapping} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (intended to be constrained by instantiations of the Web Infrastructure Model)
- $s_0^p.\texttt{sts} \equiv \langle\rangle$
- $s_0^p.\texttt{DNSaddress} \in \mathsf{IPs}$ (note that this includes the possibility of using an attacker-controlled address)
- $s_0^p.\texttt{pendingDNS} \equiv \langle\rangle$
- $s_0^p.\texttt{pendingRequests} \equiv \langle\rangle$
- $s_0^p.\texttt{isCorrupted} \equiv \bot$
- $s_0^p.\texttt{cibaBindingMessages} \equiv \langle\rangle$
- $s_0^p.\texttt{tlskeys} \equiv tlskeys^p$ (see Appendix D-C)

Note that instantiations of the Web Infrastructure Model may define different conditions for a web browser's initial state.

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 $\mathsf{CHOOSEINPUT}(s', scriptinputs)$ 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 $\mathsf{PARENTWINDOW}(tree, docnonce)$ 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 $\bot$.

d) PARENTDOCNONCE: The function $\mathsf{PARENTDOCNONCE}(tree, docnonce)$ 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 $\mathsf{SUBWINDOWS}(tree, docnonce)$ 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, $\mathsf{SUBWINDOWS}(tree, docnonce)$ returns $subwindows$.

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 $\mathsf{GETORIGIN}(tree, docnonce)$ 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.

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 $body := \langle\rangle$
    let $startciba \leftarrow \{\top, \bot\}$
    if $startciba \equiv \top$ **then**
      let $body[\texttt{authServ}] \leftarrow \mathsf{Doms}$
      let $body[\texttt{identity}] \leftarrow s'.\texttt{ids}$
    let $url := \langle\texttt{URL}, protocol, host, path, parameters, fragment\rangle$
    let $req := \langle\texttt{HTTPReq}, \nu_2, \texttt{GET}, host, path, parameters, \langle\rangle, body\rangle$
    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 $\langle\rangle, s'$

else if $\exists\, \langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **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 $\langle reference, request, url, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
  if $\texttt{binding\_message} \in^{\langle\rangle} m'.\texttt{body}$ **then**
    let $s'.\texttt{cibaBindingMessages} := s'.\texttt{cibaBindingMessages} +^{\langle\rangle} \langle request.\texttt{host}, m'.\texttt{body}[\texttt{binding\_message}]\rangle$
  call PROCESSRESPONSE(m', $reference$, $request$, url, a, f, s')
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** $m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
  remove $\langle reference, request, url, \bot, f\rangle$ **from** $s'.\texttt{pendingRequests}$
  call PROCESSRESPONSE(m, $reference$, $request$, url, a, f, s')
else if $m \in$ DNSResponses **then** → Successful DNS response
  if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin$ IPs $\vee\, m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].\texttt{request.host}$ **then**
    stop
  let $\langle reference, message, url\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
  if $url.\texttt{protocol} \equiv \texttt{S}$ **then**
    let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, message, url, \nu_3, m.\texttt{result}\rangle$
    let $message := \mathsf{enc_a}(\langle message, \nu_3\rangle, s'.\texttt{keyMapping}\,[message.\texttt{host}])$
  else
    let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
  let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
  stop $\langle\langle m.\texttt{result}, a, message\rangle\rangle$, s'
else 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**
  → For modelling CIBA, we allow the browser to receive requests. By this, the AS can contact its users and ask to give their consent for a given CIBA flow
  let n, $method$, $path$, $parameters$, $headers$, $body$ **such that** $\langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$ **if possible; otherwise stop**
  if $path \not\equiv /\texttt{start} - \texttt{ciba} - \texttt{authentication}$ **then stop**
  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 $url := body[\texttt{ciba\_url}]$
  let $req := \langle \texttt{HTTPReq}, \nu_{\text{ciba\_req}}, \texttt{POST}, url.\texttt{host}, \varepsilon, \langle\rangle, \langle\rangle, body\rangle$
  call HTTP_SEND($\langle \texttt{REQ}, windownonce\rangle$, req, url, $\bot$, $\bot$, $\bot$, a, s')
stop

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

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

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

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

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

J. DNS Servers

Definition 69. 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 70. 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 71 (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$(m_{\text{dec}}, k, a, f, s')$
else if $m \in$ DNSResponses **then** $\rightarrow$ 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** $\rightarrow$ Encrypted HTTP response
  let $m' := \mathsf{dec_s}(m, key)$
  if $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
    stop
  if $m' \notin$ HTTPResponses **then**
    call PROCESS_OTHER(m, a, f, s')
  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** $\rightarrow$ Process was triggered
  call PROCESS_TRIGGER(a, s')
else
  call PROCESS_OTHER(m, a, f, s')
stop

M. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [34].

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 72 (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 73. 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 74. 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 75. 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 76. 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 77. 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 78. 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 79. 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 80. 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 24. 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 71ff. 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 51ff. 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 77 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 24, 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 25. 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 76 (or Line 79 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 77 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 24 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 26. 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 90 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 25 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 24, but is applied to the generic HTTPS server (instead of the web browser).

Lemma 27. 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 24:

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'.